Contexts in Lambda Calculus

Bognar, M.

2002

Contexts in Lambda Calculus

Cover design by Bernards/Visser communicatie bv

VRIJE UNIVERSITEIT

Contexts in Lambda Calculus

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 26 november 2002 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Mirna Bognar

geboren te Slavonski Brod, Kroatië

promotor: prof.dr. J.W. Klop
copromotor: dr. R.C. de Vrijer

To my parents and

the memory of my father

Acknowledgements

Many people have helped me to contribute my modest share, or more precisely, my cube, to theoretical computer science. Here, I wish to express my gratitude for their help.

In the first place, I am deeply grateful to my supervisors Jan Willem Klop and Roel de Vrijer for giving me the opportunity and freedom to (re)search, and for offering their guidance in the process of writing this thesis. From my student–assistant days until the last revisions of this thesis, they have provided chances and inspiration, and most of all, believed I was up to the challenge.

I would like to thank the members of my reading committee, H.P. Barendregt, F. Kamareddine, V. van Oostrom and W.J. Fokkink, for their effort and useful comments on this thesis. I wish to thank H.P. Barendregt for his interest in my work during many years; F. Kamareddine for her enthusiasm and useful discussions about my work, and for her hospitality during the Edinburgh-sited workshops; V. van Oostrom for introducing me to higher-order rewriting and for keeping an eye on my research for all these years; and W.J. Fokkink for his interest and encouragement related to my work. I appreciate that the members of the reading committee accompanied by N.G. de Bruijn, J. van Eijck, J.H. Geuvers, T. Hardin and R.P. Nederpelt are willing to take place in my promotion committee, and I am honoured to have such a prominent opposition.

During my stay at the Vrije Universiteit many people have contributed to making the university a pleasant workplace. I enjoyed working together with my colleagues, among others, Erik de Vink and Yde Venema. My roommates through these years, Marcello Bonsangue, Jerry den Hartog, Bas Luttik and Wan Fokkink, provided welcome and fruitful distractions through many useful and unuseful discussions on, among other subjects, the Dutch language and on bluffing one's way into taking chances, and through the discovery that our phone ring actually works (at least, on Fridays). I hope the present PhD students, Paulien de Wind, Clemens Grabmayer and Jeroen Ketema, will have at least as much joy in writing their theses as I had while writing this thesis.

My lunch mates, among others, Anton Eliëns, Arno Bakker, Chris Verhoef, Frank Dehne, Frank Niessink, Gerard Kok, Gerco Ballintijn, Hans Burg, Jaap Gordijn, Michel Klein, Michel Oey, Radu Serban and Stefan Blom, have made lunch breaks into friendly meetings. A chocolate muffin is just not the same anymore.

The ICT department, especially Ruud Wiggers and Gert Huisman who I have

bothered the most, took care of all my computer problems. Gert enabled me to meet my deadlines, and I hope to keep meeting Ruud at mineral and fossil markets.

The trainers at the VU sport centre, especially Hans Smit, have motivated me to keep training by, somehow, transforming training sessions into bike rides on bendy roads over green hills. I thank Gerco, Floor and other 'seniors' for their company during these bike rides.

Outside the university, my friends and family were very much involved in my thesis saga and beyond. My friends, Michel, Gerard, Mirjana & Bart, John & Elisabeth, Maartje & Chris, Jenny & Frank, Jaap & Sanne, Kim & Jeanôt, offered an ear, shoulder or distraction, whichever whenever needed. Michel and Gerard, my 'adopted Dutch family', even agreed to act as paranymphs. I am grateful to Željka, Maja and Irena for their support through their ageless, distanceless friendship.

I thank Stan and his family for their love and support. I feel lucky for having Stan and our Maleni as my 'α, β and ω males.'

Last but by no means least, I thank my family, Mum Gordana and Dad Mirko, Boris, Martina and Vlado, for their love and for teaching me The Most Important Things in Life. My big brother Boris, thank you for teaching me how to read, write, give and take. Mum, thank you for teaching me how to love and learn, and Mirko, thank you for being proud of what I have learned.

Amsterdam, October 2002

Contents

Introduction

The background of this thesis is the formalisation and verification of mathematical reasoning, with or without the aid of a computer. This thesis presents a representation of mathematical structures as contexts in such a formalisation. The study on representation of mathematical structures as contexts has led to a broader study and formalisation of contexts.

Mathematics and mathematical logic

Mathematics is the science of expressing and studying the relationships between quantities, magnitudes and more general abstract concepts. It also embraces logical reasoning based on definitions, axioms, assumptions and rules for combining and transforming primitive notions into more complex objects, relations and theorems.

When conducting a particular mathematical argument, one assumes a comfortable level of abstraction. A comfortable level of abstraction depends on the contents and aim of the mathematical argument, on the expertise of the mathematician performing the argument and on the expertise of his audience. It is established by the choice of a collection of primitive notions, possibly from some generally accepted primitive notions, and by the choice of a granulation degree of the reasoning steps. The contents and the aim of an argument determine mainly the primitive notions, while the expertise of the parties involved determines the granulation degree of the reasoning steps.

The day–to–day mathematical practice is a human activity. Since to err is also human, mathematical proofs are prone to contain errors.

The credibility of a mathematical argument can be enhanced by recognising patterns of logical reasoning as independent from the contents of the argument and by verifying that these reasoning steps are applications of some sound logical rules. Here, a collection of logical rules are called sound if no contradiction follows from the rules.

Ultimately, one can formalise mathematical arguments as follows. First, a language of mathematical statements is agreed upon. This language formally represents natural language sentences in an unambiguous way. Next, a collection of logical axioms (statements known to be true) and rules are singled out and verified to be sound. When isolating logical axioms and rules, one preferably chooses a minimal collection, from which other necessary statements can be derived. Finally, a math-

ematical argument is translated into the formal language and it is verified that the logical steps in the argument are performed in accordance with the logical axioms and rules.

Formalisation of mathematical reasoning has been studied as early as in ancient Greece. In this thesis we will focus on the Brouwer–Heyting–Kolmogorov constructive interpretation of mathematical arguments (see for example [TD88]).

Formalisation of mathematical logic in type theories

Type theories provide a framework for formalisation of mathematical reasoning. More precisely, in type theories a deductive method of reasoning can be formalised, in which an argument is considered true if at each stage of the argument the current statement follows logically from the statements that preceded it.

Technically, type theories deal with terms, types and a typing relation which relates them. If a term M and a type τ are related by the typing relation, which is denoted by $M : \tau$, then we say that M is of type τ. The typing relation is generated from a collection of typing rules.

In type theory, mathematical statements can be represented as types and their proofs can be represented as terms. By understanding types as mathematical statements and terms as proofs, the typing relation $M : \tau$ establishes when M represents a proof of τ. In this perspective on type theories, the typing rules represent elementary logical reasoning steps. This correspondence between logic and type theories is called the propositions–as–types principle. It is justified by what is called the Curry–Howard–de Bruijn isomorphism.

In this thesis we concentrate on the type theories based on lambda calculus, which are called typed lambda calculi. There exist many typed lambda calculi, which correspond to different logics. We shall concentrate on Barendregt's lambda cube (see [Bar92]), which presents eight different typed lambda calculi in a uniform way. It may be noted that some logics can be coded into other, possibly weaker, logics; we shall not consider such codings in this thesis.

In order to give a flavour of how mathematical reasoning can be formalised within a certain typed lambda calculus, we treat here an example in the simply typed lambda calculus $\lambda^{\to}$. The simply typed lambda calculus $\lambda^{\to}$ may be employed to formalise the first-order (constructive[1]) propositional logic with one logical connective, namely the implication $\to$.

The types of $\lambda^{\to}$ are built from propositional variables $P, Q, R, \ldots$ and by forming implications $\tau \to \sigma$. The implication $\tau \to \sigma$ is interpreted as the formalisation of the natural language statement saying '*if* τ *then* σ,' or equivalently 'τ *implies* σ.' In general, the types formed depend on the typed lambda calculus used which corresponds to different logics.

In $\lambda^{\to}$, terms are formed as in typed lambda calculi, from variables $x, y, \ldots$, and by forming abstractions $\lambda x{:}\tau.\, M$ and applications $M\,N$. We will explain the interpretation of terms together with the typing rules. Here, $\lambda x{:}\tau$ is a binder which

[1] Typically, in constructive logic one cannot derive that 'P *or not* P' is true.

$$(var) \quad \Gamma \vdash x : \tau \qquad \text{if } (x : \tau) \in \Gamma$$

$$(abs) \quad \frac{\Gamma \cup \{x : \tau\} \vdash M : \sigma}{\Gamma \vdash (\lambda x{:}\tau.\, M) : \tau \to \sigma}$$

$$(app) \quad \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M\,N : \sigma}$$

Figure 1: The typing rules for $\lambda^{\to}$

binds the free occurrences of the variable x of type τ in M. This binding behaviour is comparable to the binding behaviour of the binder $\forall x \in \mathbb{N}$ in for example $\forall x \in \mathbb{N}.x \geq 0$ or of the binder $\int\ dx$ in for example $\int x dx$. In general, a specific typed lambda calculus may extend this general syntax of terms.

The typing relation $M : \tau$ is generated by the typing rules displayed in Figure 1. In the figure, Γ denotes a set of assumptions.

The typing rules are interpreted as follows. The axiom (var) states that if τ is among the assumptions, then τ is true. The proof of τ is the same as in the assumption, that is x. The rule (abs) formalises the reasoning step saying that if assuming τ, the statement σ can be derived, then $\tau \to \sigma$. The proof of $\tau \to \sigma$ is a recipe, a function, $\lambda x{:}\tau.\, M$, which takes the proof of τ as an argument and transforms it into a proof of σ. The rule (app) formalises what is called modus ponens, saying that if $\tau \to \sigma$ and if τ, then σ. The proof of σ is the application of the proof of $\tau \to \sigma$ to the proof of τ.

By using the typing rules, statements and their proofs can be derived. A derivation is technically a 'tree' of statements where each statement is either an axiom or it follows from the statements immediately above it by some rule.

Example. The following derivation can be built with $\Gamma = \{p : P, a_1 : P{\to}Q, a_2 : Q{\to}R\}$:

$$\frac{\dfrac{\Gamma \cup \{a_3 : P{\to}R\} \vdash a_3 : P{\to}R \qquad \Gamma \cup \{a_3 : P{\to}R\} \vdash p : P}{\Gamma \cup \{a_3 : P{\to}R\} \vdash a_3 p : R}}{\Gamma \vdash (\lambda a_3{:}P{\to}R.\, a_3 p) : (P{\to}R){\to}R.}$$

This derivation of a typing statement represents the following mathematical argument. The elements of Γ represent the assumptions: we assume that P, $P{\to}Q$ and $Q{\to}R$ are true, or more precisely, we assume that there are proofs p, a_1 and a_2 of respectively P, $P{\to}Q$ and $Q{\to}R$. The type $(P{\to}R){\to}R$ represents the statement saying if P implies R, then R. The term $\lambda a_3{:}P{\to}R.\, a_3 p$ represents the proof of the statement and it shows how the proof of R can be constructed from the proofs p, a_1, and a_2.

There is also a computational aspect in typed lambda calculi. The principle of computation is called β-reduction and it is generated by

$$(\lambda x{:}\tau.\, M)\, N \to_\beta M\,[\![x := N]\!] \qquad (\beta)$$

where $M[\![x := N]\!]$ denotes the result of substituting N for x in M. Its interpretation depends on the interpretation of the terms involved. In the line of the example above given for the simply typed lambda calculus, such computation steps arise from the detours in a proof. Detours correspond to introducing abbreviations in informal mathematics, which contributes to the structure and understanding of an argument.

We show this by an example.

Example. In the example above, we can in fact also deduce the 'transitivity lemma' saying that, if P implies Q and if Q implies R, then P implies R. That is, there is a derivation ending in

$$\Gamma \;\vdash\; (\lambda p'{:}P.\, a_2\,(a_1\,p')) : P{\to}R.$$

This lemma can be used in the proof $\lambda a_3{:}P{\to}R.\, a_3\,p$ of the previous example instead of the additional assumption $a_3 : P{\to}R$. The usage of the lemma corresponds to applying the proof of $(P{\to}R){\to}R$ to the proof of $P{\to}R$, as is shown in this last fragment of the derivation:

$$\frac{\Gamma \;\vdash\; (\lambda a_3{:}P{\to}R.\, a_3\,p) : (P{\to}R){\to}R \qquad \Gamma \;\vdash\; (\lambda p'{:}P.\, a_2\,(a_1\,p')) : P{\to}R}{\Gamma \;\vdash\; (\lambda a_3{:}P{\to}R.\, a_3\,p)\,(\lambda p'{:}P.\, a_2\,(a_1\,p')) : R.}$$

The term $(\lambda a_3{:}P{\to}R.\, a_3\,p)\,(\lambda p'{:}P.\, a_2\,(a_1\,p'))$ computes to $(\lambda p'{:}P.\, a_2\,(a_1\,p'))\,p$, which in turn computes to $a_2\,(a_1\,p)$, which represents a direct proof of R from Γ.

An additional technical advantage of introducing detours, which is here not clearly visible due to the simplicity of the example, is that proofs in general become smaller in size: if N is large, and if x is used in M many times, then a proof $(\lambda x{:}\tau.\, M)\,N$ with a detour is smaller than a more direct proof $M[\![x := N]\!]$. Hence, in a proof $(\lambda x{:}\tau.\, M)\,N$, the application–abstraction construction $(\lambda x{:}\tau.\, \_)\,N$ can be understood as an abbreviation mechanism, where x abbreviates N in M.

Formalisation of mathematics in type theories

The ideal of formalisation of mathematics is to bring formal reasoning as close as possible to the informal mathematical practice.

From the point of view of mathematical logic, the expressive power of the simply typed lambda calculus is rather limited. An average mathematical argument uses a more expressive logic than propositional logic. Choosing a suitable typed lambda calculus, which corresponds to a proper logic is one of the methods to accomplish this ideal.

Furthermore, mathematics is not only logic. Again, an average mathematical argument deals not only with primitive notions such as proposition variables but

also with diverse and complex structures such as lists and algebras. Moreover, when conducting an argument a mathematician employs definitions, additional global axioms, etc.

We name a few methods for enhancing the formal exposition of mathematics.

By considering also sets $A, B, \ldots$ functions and predicates can be represented too. Then, for example, the term $\lambda x{:}A.\, \lambda y{:}B.\, x$ of type $A{\rightarrow}(B{\rightarrow}A)$ represents a function, R of type $A{\rightarrow}A{\rightarrow}\mathit{Prop}$ represents a binary relation over the set A and *ref* of type $\forall x : A.Rxx$ represents the property that R is reflexive.

By adding new global axioms or principles one can capture, for example, classical[2] logic, or reason with inductive types (see for example [CPM90, PPM90]). In the presence of inductive types, one can also define objects by induction, perform proofs by induction on their structure and define functions by recursion based on objects that are inductively defined.

By refining the notion of computation in the logical framework one can support definitions (see [SP94]), or capture stepwise computation of substitutions. The latter yields calculi with explicit substitutions (see for example [Blo99]), which are in particular interesting for implementing typed lambda calculi, because in implementations substitution is indeed computed stepwise.

In any case, the logical rules of an extended or refined typed lambda calculus, should of course be sound.

In general, formalised mathematical arguments tend to grow in length and meticulous details, but they gain in structure and credibility.

Automated reasoning

The nature of verifying the reasoning steps of an argument as represented in a typed lambda calculus suggests that the verification can be performed mechanically. The advent of computers has given the formalisation of mathematics an impulse by providing a perfect mechanical verifier of tediously rigorous expositions of proofs.

In 1966 N.G. de Bruijn started the Automath project with the goal of designing 'a language for expressing a very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of grammar are obeyed' (see [Bru70]), and ultimately, with the goal of implementing a program to verify mathematical texts written in this language by computer. The project resulted in a family of languages and the implementations of some of them.

Today, there is a vast range of tools for automated reasoning. They support verification, assistance in finding proofs or even, in some simpler cases, finding proofs automatically. These systems are not only tools based on type theories, but they also implement other mathematics formalisation techniques. We name but a few tools: Coq (see [Coq]), HOL (see for example [HOL]), Isabelle (see for example [Isa]), LEGO (see [LEG]), Mizar (see for example [Miz]), Nuprl (see [Nup]), PVS (see [PVS]), and Yarrow (see [Yar]).

[2] Typically, in classical logic one can derive that '*P or not P*' is true.

Contexts

Contexts play a role not only in lambda calculus but also in many other systems of expressions and expression transformation, like for example in programming languages, and in linguistics. In general, contexts are expressions with special places, called holes, where other expressions may be placed. For example, in the lambda calculus, $(\lambda x{:}\tau.\,[])\,z$ where [] denotes a hole, is a context. The operation of literally replacing the holes of a context by expressions is called hole filling and it is usually denoted by [ ]. For example, filling the term $x\,z$ into the hole of the context above results in the term $(\lambda x{:}\tau.\,x\,z)\,z$. In formal systems with binders, such as lambda calculus, a distinctive feature of hole filling is variable capturing: some free variables of the expression placed into a hole of a context may become bound by the binders of the context. In the example, the free variable x of the term $x\,z$ has become bound by the binder λx of the context. On this point, hole filling differs from substitution, where variable capturing is avoided. For example, if [] were considered as a variable, the substitution of $x\,z$ for [] in $(\lambda x{:}\tau.\,[])\,z$ would result in $(\lambda x'{:}\tau.\,x\,z)\,z$, where λx has been renamed to avoid capturing of x in $x\,z$.

In many formal systems, the standard transformations which are defined on expressions, are not defined on contexts. That implies that contexts are treated merely as a notation, which hinders any formal reasoning about or with contexts.

Formalisation of contexts is motivated in different realms of research as diverse as proof checking, programming languages, operational semantics and natural languages. For instance, contexts are formalised with the purpose of optimisation of interactive proof checking (see [Mag96]), representing incomplete proofs and supporting incremental proof development (see [Muñ97, GJ02]), developing programming languages (see [HO98, SSB99, LF96]), dealing with contexts in operational semantics (see [San98, Mas99]), and modelling binding mechanisms in natural language (see [KKM99]).

In this thesis, we consider one application of contexts in particular, namely the application of contexts in representation of mathematical structures, which can in turn be employed in proof checking.

Formalisation of mathematical structures as contexts

An essential part in mathematics are structures collecting sets, relations, functions and assumptions. Examples of such structures are algebras and relations. Algebras collect sets and operations on them satisfying certain properties, and relations consist of sets, a relation on the sets and its properties. Such mathematical structures can be represented as contexts in a typed lambda calculus.

For example, a reflexive binary relation, which consists of a set, a binary relation and the assumption that the relation is reflexive, can be represented by

$$\lambda S{:}\mathit{Set}.\,\lambda R{:}S{\to}S{\to}\mathit{Prop}.\,\lambda r{:}(\forall x{:}S.\mathit{Rxx}).\,[].$$

Let us call this context $\underline{\mathit{re\_ref}}$. Then, an argument on reflexive relations, say a piece

of mathematical text *text*, can be performed within this context, via hole filling:

$\underline{re\_ref}[text]$.

In *text* the identifiers S, R, r can then be used.

The idea to represent such structures as contexts is due to N.G. de Bruijn. Contexts representing mathematical structures were called segments, because of the particular structure of the contexts (which is a bit more general than the example above suggests). Some work on formalisation of this idea has been done by H. Balsters (see [Bal86, Bal87]), I. Zandleven and N.G. de Bruijn (see [Bru91]), but these formalisms are not sufficient for direct representation of mathematical structures, which requires lambda calculi with dependent types.

Subtyping

In informal mathematics, a mathematician often makes larger reasoning steps than the reasoning steps of formal mathematics. One of such steps involves reasoning about structures $\mathcal{A}$ and $\mathcal{B}$ for which one may say that 'every $\mathcal{A}$ is a $\mathcal{B}$'. Consider the example of reflexive relations and equivalence relations. One may say that an equivalence relation E is also a reflexive relation because E is also a binary relation over a set and its properties imply reflexivity. Then each lemma about a reflexive relation can be applied to an equivalence relation. This kind of reasoning is said to involve subtyping. In the present setting, a direct application of such a lemma is not possible; one has to reproduce the proof for an equivalence relation explicitly.

With mathematical structures represented as contexts, we believe that reasoning with subtyping can be formalised internally.

Contributions of this thesis

The main contributions of this thesis are:

- a framework for formalisations of contexts;
- a definition of a notion of context for the systems of the lambda cube;
- a formalisation of mathematical structures in a lambda calculus with dependent types; and,
- an idea of how subtyping could be defined on the representations of mathematical structures.

We design two context formalisms, namely the context calculus and the context cube. The latter is a collection of eight context calculi which are related to the systems of Barendregt's lambda cube with contexts. Both the context calculus and the context cube employ basically the same approach to the formalisation of contexts, but they differ in their expressivity.

In the context calculus the emphasis is placed on illustrating the flexibility of our approach to the formalisation of contexts. In the context calculus, one can

represent different notions of context and different notions of context formalisation. We will show how contexts with many holes and segments can be represented, and how functions ranging over contexts can be allowed or omitted in a context formalisation. In the context calculus, the expressivity related to the type system will be limited and the holes will be allowed to occur only in terms.

In the context cube the emphasis is put on the expressive power of our approach to context formalisation. The context cube includes type systems which correspond to logics expressive enough for encoding mathematical structures.

Both context formalisms are extensions of lambda calculus, which can be translated back into lambda calculus. Hence, the context formalisms can be conceived as a comfortable level of abstraction within the lambda calculus for dealing with contexts.

With a powerful logical framework at our disposal, we implement the idea of N.G. de Bruijn, and represent mathematical algebras as contexts. Once mathematical algebras are represented they can be freely manipulated. One can define bigger structures, functions ranging over mathematical structures and theorems involving mathematical structures. That is, one can reason about mathematical structures in a formal mathematical framework as in informal mathematics. The fact that the context cube can be translated into the lambda cube, implies that this formalisation of mathematical structures can be used in the existing tools for automated reasoning that are based on the lambda cube or type theories.

Overview

This thesis is organised as follows. The heart of the thesis is Chapter 2, which presents an introduction to contexts in lambda calculus, motivates formalisation of contexts, and gently introduces our approach to formalisation of contexts. Chapters 3, 4, 5 and 6 present the implementations of our approach. Chapters 3, 4 and 5 present the context calculus and its applications. Chapter 6 presents the context cube and its applications. Chapter 7 indicates a possible extension of our approach with subtyping.

We discuss the chapters separately.

Chapter 1 is a preliminary chapter about rewriting. Rewriting plays an important role in this thesis because rewriting techniques will be used to model computation with contexts. This chapter is an introduction to different rewrite systems, properties of rewrite systems and results. It covers abstract rewriting, which provides the basics of the rewriting theory for this thesis; lambda calculus, whose contexts we will formalise; and higher-order rewriting, which provides results that will be used throughout the thesis. Moreover, this chapter conveys the terminology and notation of the thesis.

Chapter 2 is an introduction to the contexts of lambda calculus. This chapter collects the notions of lambda context that are encountered in the literature on rewriting. Here, formalisation of contexts in general is explained, discussed and motivated. Also, different existing formalisations of contexts are considered and compared. This chapter also explains our approach to formalisation of lambda

contexts in the context calculus and the context cube.

Chapter 3 contains the definition of the context calculus and investigates the properties of rewriting in the context calculus. It discusses the techniques used in the formalisation of contexts and shows that this approach can be used also in other rewrite systems, with or without binders.

Chapters 4 and 5 present four examples of applications of the context calculus. These examples are defined via typing, which guards the formation of expressions. The examples illustrate different modes of formalisation and different notions of context which can be captured by the context calculus. In particular these examples formalise the following:

- the untyped lambda calculus with contexts: this application repairs the shortcomings of the naïve way of reasoning with contexts in the lambda calculus, and defines β-reduction also on (representations of) contexts;
- the simply typed lambda calculus with contexts: this application includes functions ranging over contexts;
- the untyped lambda calculus with contexts where also functions ranging over contexts are defined; and
- the simply typed lambda calculus with de Bruijn's segments.

In all these examples, it is proved that the stepwise computation related to contexts is strongly normalising (i.e. each context-related computation yields a result) and confluent (which guarantees that results of computations are unique).

In Chapter 6 our approach to formalisation of contexts is extended to the lambda calculi with dependent types. The lambda calculi with dependent types have greater expressive power than for example the simply typed lambda calculus, but dependent types introduce more technical complexities. In this chapter we present the context cube, which is a collection of eight systems corresponding to the eight systems of Barendregt's lambda cube with contexts. It is shown that the context cube can be used for representation of mathematical structures in a uniform way. In these systems, the stepwise computation is strongly normalising and confluent. The importance of the context cube lies in its application in proof checking, and in the fact that it indirectly defines a notion of context for the lambda cube.

Chapter 7 concludes this thesis by setting out two ideas for future work. The first idea concerns improving the communication mechanism between contexts and expressions to be put into their holes by adding labels. The second idea outlines how a notion of subtyping on representations of mathematical structures can be defined by using labels.

Chapters 3, 4 and 5 are based on the work presented in [BV01] and [BV99]; Chapter 6 is based on [BV02].

Chapter 1

Introduction to rewriting

In this thesis we focus on computation with contexts. We will formalise this type of computation by using rewrite systems, which provide a standard method for modelling computation. In this chapter an introduction to rewrite systems is provided.

This chapter is a preliminary one. It collects definitions and well-known results that will be used throughout the thesis. This chapter contains no original material, except for a more general definition (Definition 1.1.14) of a subsystem of an abstract reduction system than the standard one (cf. [Klo92]).

In a rewrite system objects involved in computation are represented and a stepwise computation is modelled by rewrite steps between representations of objects. The exact form of object representations and the degree of the computation granulation depend on the aspects of computation that one aims to model and on the properties of computation that one intends to study. While the computation aspects of interest may vary considerably, the desirable properties of computation are in many cases the same. These include the properties of the existence and uniqueness of results, which are the most natural requirements for computation: starting from a given object, some or every computation has a result, and if there is a result, then it is unique.

This chapter is organised as follows. Section 1.1 is about abstract rewriting, which models computation as a relation between objects whose structure is considered abstract. In this section general notions, terminology and properties regarding computation are given. Section 1.2 is about lambda calculus, which is an example of a rewrite system where one studies how computation changes the structure of an object, in order to get more grip on the properties that the rewrite system has. In this section, terminology is more specific and local tests are formulated that imply the desirable properties mentioned above. Section 1.2.1 deals with the untyped lambda calculus and Sections 1.2.2 and 1.2.3 deal with typed lambda calculi, where the structure of terms is restricted by typing. Section 1.3 considers higher-order rewrite systems, a framework for arbitrary term rewrite systems, in which the structure of objects and of rewrite steps can be presented in a uniform way. Last but not least, Section 1.4 lists some notational abbreviations and conventions that will be used

throughout the thesis.

1.1 Abstract rewriting

The material and notation presented in this section is based on and, for the most part, agrees with [Klo92]. For more information on abstract rewriting, see also [Bog95].

Abstract reduction systems model computation by abstracting from the object structure and by concentrating on interaction of computation steps. When computation steps are associated with computations of different nature, the computation steps may be indexed.

Let I be a countably infinite set of indices.

Definition 1.1.1 (Abstract reduction system (ARS)) An abstract reduction system is a structure $\mathcal{A} = \langle A, (\to_\imath)_{\imath \in I} \rangle$ consisting of a set of objects A, and a collection of binary relations $\to_\imath$ on A, indexed by a set I. For $\imath \in I$, the relations $\to_\imath$ are called rewrite or reduction relations. We will often refer to the relation $\to_\imath$ as the $\imath$-rewrite relation.

The next definition involves relations between objects of an ARS. In fact, the definition formulates a couple of binary relations on a set, and as such it could have been given in a more general form than in the context of abstract rewriting. We prefer the definition in the form as given, because it is all we will need throughout the thesis.

Definition 1.1.2 Let $\mathcal{A} = \langle A, (\to_\imath)_{\imath \in I} \rangle$ be an ARS.

i) The identity relation, denoted by $\equiv$, on the elements of A is defined by

$$\equiv = \{(a, a) \mid a \in A\}.$$

ii) Let $\imath, \jmath \in I$. The union of $\to_\imath$ and $\to_\jmath$, denoted by $\to_{\imath\jmath}$, is the binary relation on A defined by

$$\to_{\imath\jmath} = \{(a, b) \in A \times A \mid (a, b) \in \to_\imath \text{ or } (a, b) \in \to_\jmath\}.$$

iii) Let $\imath \in I$. The inverse of $\to_\imath$, denoted by $\leftarrow_\imath$, is the binary relation on A defined by

$$\leftarrow_\imath = \{(b, a) \in A \times A \mid (a, b) \in \to_\imath\}.$$

iv) Let $\imath, \jmath \in I$. The sequential composition of $\to_\imath$ and $\to_\jmath$, denoted by $\to_\imath ; \to_\jmath$, is the binary relation on A defined by

$$\to_\imath ; \to_\jmath = \{(a, c) \in A \times A \mid \exists b \in A \text{ such that } (a, b) \in \to_\imath \text{ and } (b, c) \in \to_\jmath\}.$$

v) Let $\imath \in I$. The n-fold composition of $\rightarrow_\imath$, denoted by $\rightarrow_\imath^n$, is the binary relation on A defined inductively by

$$\begin{aligned} \rightarrow_\imath^0 &= \equiv \\ \rightarrow_\imath^{n+1} &= \rightarrow_\imath^n ; \rightarrow_\imath \quad \text{for } n \geq 0. \end{aligned}$$

Notation. If I is a singleton, then the indices are left out.

The next definition is about different closures of a rewrite relation. In this definition, given a property P of a relation, the rewrite relation is extended in such a way that the extension satisfies the property P and that it is minimal with respect to the ordering on sets.

Definition 1.1.3 Let $\mathcal{A} = \langle A, \rightarrow \rangle$ be an ARS.

i) The reflexive closure of $\rightarrow$, denoted by $\rightarrow^{\equiv}$, is the smallest extension of $\rightarrow$ such that $\rightarrow^{\equiv}$ is reflexive.

ii) The symmetric closure of $\rightarrow$, denoted by $\leftrightarrow$, is the smallest extension of $\rightarrow$ such that $\leftrightarrow$ is symmetric.

iii) The transitive closure of $\rightarrow$, denoted by $\rightarrow^+$, is the smallest extension of $\rightarrow$ such that $\rightarrow^+$ is transitive.

iv) The reflexive–transitive closure of $\rightarrow$, denoted by $\twoheadrightarrow$, is the smallest extension of $\rightarrow$ such that $\twoheadrightarrow$ is both reflexive and transitive.

v) The equivalence closure of $\rightarrow$, also called the convertibility relation generated by $\rightarrow$ and denoted by $=$, is defined as the reflexive–transitive closure of the symmetric closure of $\rightarrow$.

vi) In an analogous way, the same kind of closures are defined for $\leftarrow$, the inverse of $\rightarrow$. The closures are also denoted in an analogous way; in particular, the reflexive–transitive closure of $\leftarrow$ is denoted by $\twoheadleftarrow$.

Notation. The rewrite relations are traditionally used in infix notation. For example, instead of $(a, b) \in \rightarrow_\imath$ we write $a \rightarrow_\imath b$.

We list some rewriting terminology in an ARS $\mathcal{A} = \langle A, (\rightarrow_\imath)_{\imath \in I} \rangle$.

Let $a \rightarrow_\imath b$. We call this an *(*$\imath$*-)rewrite step*. In this rewrite step a is a *(*$\imath$*-)redex* and b is a *one-step (*$\imath$*-)reduct* of a.

Let $a_0 \rightarrow_\imath \ldots \rightarrow_\imath a_n$ where $n \geq 0$. We call $a_0, \ldots, a_n$ an ($\imath$-)rewrite sequence from a_0 to a_n. We also say that a_0 *reduces* or *rewrites* to a_n and that a_0 is an *(*$\imath$*-)reduct* of a_0. An empty rewrite sequence is denoted by ϵ.

One can prove that $a \twoheadrightarrow_\imath b$ if and only if there is an ($\imath$-)rewrite sequence from a to b. We will write $r : a \twoheadrightarrow_\imath b$ to denote a specific rewrite sequence from a to b. However, we will often confuse $a \twoheadrightarrow b$ and a rewrite sequence from a to b.

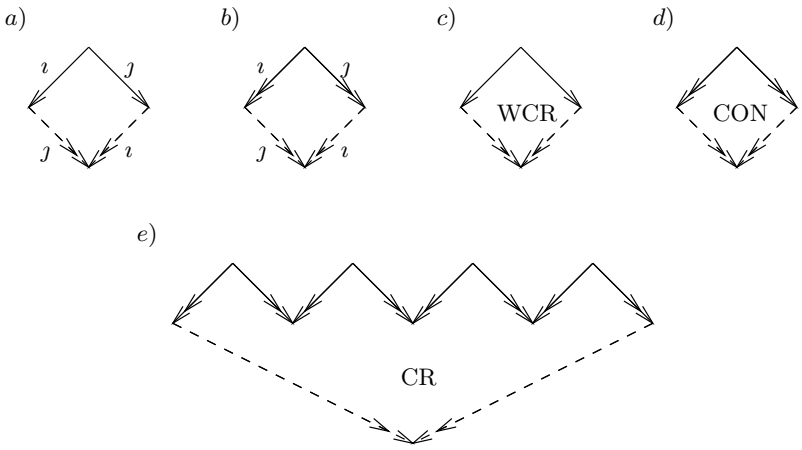


Figure 1.1: Properties of an ARS

Let $a_0 \leftrightarrow_\imath a_1 \leftrightarrow_\imath \ldots \leftrightarrow_\imath a_n$ with $n \geq 0$. We call $a_0, \ldots a_n$ an *($\imath$-)conversion* and say that the two objects a_0 and a_n are *($\imath$-)convertible*.

One can prove that $a =_\imath b$ if and only if there is an ($\imath$-)conversion between a_0 and a_n.

We will often confuse $a = b$ and a conversion between a and b. Let $a \in A$. We call a an *($\imath$-)normal form* if there is no $b \in A$ such that $a \to_\imath b$. We say that a *has a normal form* if there is an object $a' \in A$ such that $a \twoheadrightarrow a'$ and a' is a normal form.

Let $a \twoheadrightarrow b$ and $a' \twoheadrightarrow b'$ be a pair of rewrite sequences. We call $a \twoheadrightarrow b$ and $a' \twoheadrightarrow b'$ *diverging* (or coinitial) if the sequences start at the same object, that is, if $a \equiv a'$. We call $a \twoheadrightarrow b$ and $a' \twoheadrightarrow b'$ *converging* (or cofinal) if the sequences end in the same object, that is, if $b \equiv b'$.

We give an overview of properties of rewrite relations, grouped into three definitions. Definition 1.1.4 deals with the interaction of individual rewrite relations. Definition 1.1.5 deals with properties related to existence of a common reduct of convertible objects. Definition 1.1.6 deals with normal forms. Some of the properties are illustrated in Figure 1.1.

Definition 1.1.4 Let $\mathcal{A} = \langle A, (\to_\imath)_{\imath \in I} \rangle$ be an ARS.

i) For $\imath, \jmath \in I$, the relation $\imath$ commutes weakly with $\jmath$ if and only if for all $a, b, c \in A$ there is $d \in A$ such that if $b \leftarrow_\imath a \to_\jmath c$ then $b \twoheadrightarrow_\jmath d \twoheadleftarrow_\imath c$ (see Figure 1.1(a)).

ii) For $\imath, \jmath \in I$, the relation $\imath$ commutes with $\jmath$ if and only if $\twoheadrightarrow_\imath$ commutes weakly with $\twoheadrightarrow_\jmath$ (see Figure 1.1(b)).

Definition 1.1.5 Let $\mathcal{A} = \langle A, \to \rangle$ be an ARS.

i) The rewrite relation $\rightarrow$ is weakly confluent (notation WCR($\rightarrow$)) if and only if $\rightarrow$ is weakly self-commuting, that is, if and only if for all $a, b, c \in A$ there is $d \in A$ such that $b \leftarrow a \rightarrow c$ implies $b \twoheadrightarrow d \twoheadleftarrow c$ (see Figure 1.1(c)).

ii) The rewrite relation $\rightarrow$ is confluent (notation CON($\rightarrow$)) if and only if $\rightarrow$ is self-commuting, that is, if and only if for all $a, b, c \in A$ there is $d \in A$ such that $b \twoheadleftarrow a \twoheadrightarrow c$ implies $b \twoheadrightarrow d \twoheadleftarrow c$ (see Figure 1.1(d)).

iii) The rewrite relation $\rightarrow$ has the Church–Rosser property (notation CR($\rightarrow$)) if and only if each pair of convertible objects has a common reduct; that is, if and only if for all $a, b \in A$ there is $d \in A$ such that if $a = b$ then $a \twoheadrightarrow d \twoheadleftarrow b$ (see Figure 1.1(e)).

Definition 1.1.6 Let $\mathcal{A} = \langle A, \rightarrow \rangle$ be an ARS.

i) The rewrite relation $\rightarrow$ has the unique normal form property (notation UN($\rightarrow$)) if and only if convertible normal forms are identical; that is, if and only if for all $a, b \in A$ if $a = b$ and a and b are normal forms then $a \equiv b$.

ii) The rewrite relation $\rightarrow$ is weakly normalising (notation WN($\rightarrow$)) if and only if each object of A has a normal form.

iii) The rewrite relation $\rightarrow$ is strongly normalising (notation SN($\rightarrow$)) if and only if there are no infinite rewrite sequences $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots$ (∞) in $\mathcal{A}$.

A rewrite relation that is both confluent and strongly normalising is called complete.

If the rewrite relation $\rightarrow$ of $\mathcal{A} = \langle A, \rightarrow \rangle$ has the property P, we also say that $\mathcal{A}$ has the property P.

The unique normal form property and the normalisation properties are natural requirements for a computation. If a rewrite relation has the unique normal form property and the strong normalisation property, then it behaves like a function. The rest of the properties are interesting because of what they entail (like for example the confluence property), and/or because they are formulated in terms of a local test (like for example the weak confluence property). We formulate some well-known results in a couple of lemmas.

Lemma 1.1.7 *Let $\mathcal{A} = \langle A, \rightarrow \rangle$ be an ARS. Then, if SN($\rightarrow$) then WN($\rightarrow$).*

Lemma 1.1.8 *Let $\mathcal{A} = \langle A, \rightarrow \rangle$ be an ARS. If CON($\rightarrow$) then UN($\rightarrow$).*

Notation. If a rewrite relation $\rightarrow$ has the unique normal form property, then we will denote the normal form of object a with respect to $\rightarrow$ by $a{\downarrow}$.

Lemma 1.1.9 *Let $\mathcal{A} = \langle A, \rightarrow \rangle$ be an ARS. Then, CR($\rightarrow$) if and only if CON($\rightarrow$).*

Lemma 1.1.10 (Hindley–Rosen) *Let $\mathcal{A} = \langle A, (\to_\imath)_{\imath\in I}\rangle$ be an ARS such that $\to_\imath$ commutes with $\to_\jmath$ for every $\imath, \jmath \in I$. Then the union $\to = \bigcup_{\imath\in I} \to_\imath$ is confluent.*

Lemma 1.1.11 (Newman) *Let $\mathcal{A} = \langle A, \to\rangle$ be an ARS. If WCR($\to$) and SN($\to$) then CR($\to$).*

We mention a corollary of the Hindley–Rosen lemma.

Corollary 1.1.12 *Let $\mathcal{A} = \langle A, (\to_\imath)_{\imath\in I}\rangle$ be an ARS such that $\to_\imath$ commutes with $\to_\jmath$ for every $\imath, \jmath \in I$. Let $J_1, J_2 \subseteq I$, let $\to_1 = \bigcup_{\imath\in J_1} \to_\imath$ and let $\to_2 = \bigcup_{\imath\in J_2} \to_\imath$. Let $\mathcal{B} = \langle A, \to_1, \to_2\rangle$. Then $\to_1$ commutes with $\to_2$.*

Throughout this thesis, we will be interested in substructures of rewrite systems. These substructures will be of two kinds: (full) sub-ARSs, which are a standard notion in rewriting, and a new notion of indexed sub-ARSs, which we introduce here. An indexed sub-ARS is a generalised notion of a sub-ARS: the notion of sub-ARS is defined on ARSs with one rewrite relation whereas the notion of indexed sub-ARS is defined on ARSs with many (indexed) rewrite relations.

Definition 1.1.13 (Sub-ARS) Let $\mathcal{A} = \langle A, \to\rangle$ and $\mathcal{B} = \langle B, \rightsquigarrow\rangle$ be two ARSs. Then $\mathcal{B}$ is a sub-ARS of $\mathcal{A}$, denoted by $\mathcal{B} \subseteq \mathcal{A}$, if the following holds:

i) $B \subseteq A$,

ii) $\rightsquigarrow$ is the restriction of $\to$ to B, that is, $\forall b, b' \in B\ (b \rightsquigarrow b' \ \Leftrightarrow\ b \to b')$.

iii) B is closed under $\to$, that is, $\forall b \in B$ if $b \to a$ then $a \in B$.

Definition 1.1.14 (Indexed sub-ARS) Suppose $J \subseteq I$. Let $\mathcal{A} = \langle A, (\to_\imath)_{\imath\in I}\rangle$ and $\mathcal{B} = \langle B, (\rightsquigarrow_\imath)_{\imath\in J}\rangle$ be two ARSs. Then $\mathcal{B}$ is an indexed sub-ARS of $\mathcal{A}$, denoted by $\mathcal{B} \sqsubseteq \mathcal{A}$, if the following holds:

i) $B \subseteq A$,

ii) for each $\imath \in J$ it holds that $\rightsquigarrow_\imath$ is the restriction of $\to_\imath$ to B, that is, $\forall b, b' \in B\ (b \rightsquigarrow_\imath b' \ \Leftrightarrow\ b \to_\imath b')$.

iii) for each $\imath \in J$ it holds that B is closed under $\to_\imath$, that is, $\forall b \in B$ if $b \to_\imath a$ then $a \in B$.

We compare the two notions of subsystem. The notions of (full) sub-ARS and indexed sub-ARS coincide on ARSs with only one rewrite relation. That is, if the indices may be dropped, then $\mathcal{B} \subseteq \mathcal{A}$ if and only if $\mathcal{B} \sqsubseteq \mathcal{A}$. The indices may be ignored if both J and I are singletons, or if $I = J$ and we are not interested in indices.

Otherwise, the two notions can be compared by considering $\mathcal{A}$ and $\mathcal{B}$ as unions of ARS with only one rewrite relation. That is, let $\mathcal{A}_\imath = \langle A, \to_\imath\rangle$ for $\imath \in I$ and let

$\mathcal{B}_\imath = \langle B, \rightsquigarrow_\imath \rangle$ for $\imath \in J$; then the two notions can be compared by considering $\mathcal{A}$ and $\mathcal{B}$ as $\mathcal{A} = \bigcup_{\imath \in I} \mathcal{A}_\imath$ and $\mathcal{B} = \bigcup_{\imath \in J} \mathcal{B}_\imath$. Informally, $\mathcal{B} \sqsubseteq \mathcal{A}$ if and only if $\mathcal{B}$ is a subset of the sub-ARSs of $\mathcal{A}$. This is stated by the next lemma.

Lemma 1.1.15 *Let $\mathcal{A} = \langle A, (\rightarrow_\imath)_{\imath \in I} \rangle$ and $\mathcal{B} = \langle B, (\rightsquigarrow_\imath)_{\imath \in J} \rangle$ be two ARSs with $J \subseteq I$. Let $\mathcal{A}_\imath = \langle A, \rightarrow_\imath \rangle$ for each $\imath \in I$ and let $\mathcal{B}_\imath = \langle B, \rightsquigarrow_\imath \rangle$ for each $\imath \in J$. Then, $\mathcal{B} \sqsubseteq \mathcal{A}$ if and only if $\mathcal{B}_\imath \subseteq \mathcal{A}_\imath$ for each $\imath \in J$.*

The properties of $\mathcal{B}$ follow from the properties of its components $\mathcal{B}_\imath$ and the properties of their interaction. In turn, the properties of the components $\mathcal{B}_\imath$ may follow from the properties of $\mathcal{A}_\imath$'s. The properties WCR, CR, CON, UN, SN, and WN are 'preserved downwards' with respect to $\subseteq$ on the components of the unions, that is, if $\mathcal{A}_\imath$ has one of these properties, $\mathcal{B}_\imath$ has it too, for $\imath \in J$. Furthermore, if $\mathcal{B} \sqsubseteq \mathcal{A}$ and if each pair of rewrite relations commute with each other in $\mathcal{A}$, then so do the pairs of rewrite relations in $\mathcal{B}$. This 'downward preservation' of the commutation property of the pairs of rewrite relations is stated by the following lemma.

Lemma 1.1.16 *Let $J \subseteq I$. Let $\mathcal{A} = \langle A, (\rightarrow_\imath)_{\imath \in I} \rangle$ be an ARS such that $\rightarrow_\imath$ commutes with $\rightarrow_\jmath$ for every $\imath, \jmath \in I$. Let $\mathcal{B} = \langle B, (\rightsquigarrow_\imath)_{\imath \in J} \rangle$ be an indexed sub-ARS of $\mathcal{A}$. Then, $\rightsquigarrow_\imath$ commutes with $\rightsquigarrow_\jmath$ for every $\imath, \jmath \in J$.*

Proof: Let $\imath, \jmath \in J$, $b \in B$, and let $b \rightsquigarrow_\imath \ldots \rightsquigarrow_\imath b'$ and $b \rightsquigarrow_\jmath \ldots \rightsquigarrow_\jmath b''$ be two diverging rewrite sequences in $\mathcal{B}$. Because $\rightsquigarrow_\imath$ and $\rightsquigarrow_\jmath$ are the restrictions of $\rightarrow_\imath$ and $\rightarrow_\jmath$ to B respectively, these rewrite sequences are also rewrite sequences of $\mathcal{A}$, that is, $b \twoheadrightarrow_\imath b'$ and $b \twoheadrightarrow_\jmath b''$. Because in $\mathcal{A}$, the rewrite relations $\rightarrow_\imath$ and $\rightarrow_\jmath$ commute, there is $d \in A$ such that $b' \twoheadrightarrow_\jmath d$ and $b'' \twoheadrightarrow_\imath d$. Because these converging rewrite sequences start at elements of B, and B is closed under $\rightarrow_\imath$ and $\rightarrow_\jmath$, all elements in the converging rewrite sequences, including d, are elements of B. Because $\rightsquigarrow_\imath$ and $\rightsquigarrow_\jmath$ are the restrictions of $\rightarrow_\imath$ and $\rightarrow_\jmath$ to B, respectively, the converging rewrite sequences are in $\mathcal{B}$, that is, $b' \rightsquigarrow_\jmath d$ and $b'' \rightsquigarrow_\imath d$. QED

Example 1.1.17 In the next section, a natural example of ARSs $\mathcal{B}$ and $\mathcal{A}$ with $\mathcal{B} \sqsubseteq \mathcal{A}$ will be given: the lambda calculi λ and λ_η, for which it holds $\lambda \sqsubseteq \lambda_\eta$. The confluence of the rewriting generated by $\rightarrow_{\beta\eta}$ (Theorem 1.2.21) is proved by showing that $\rightarrow_\beta$ and $\rightarrow_\eta$ are confluent and that $\rightarrow_\beta$ commutes with $\rightarrow_\eta$ (Theorems 1.2.18 and 1.2.19 and Lemma 1.2.20).

Example 1.1.18 This example shows that an indexed sub-ARS $\mathcal{B}$ of a confluent ARS $\mathcal{A}$ may not be confluent itself. Consider the ARSs (see Figure 1.2):

$$\mathcal{A} = \langle A = \{a, b, c, d\}, \rightarrow_\imath = \{(a, b), (a, c), (c, d)\}, \rightarrow_\jmath = \{(b, d)\} \rangle$$
$$\mathcal{B} = \langle B = \{a, b, c, d\}, \rightsquigarrow_\imath = \{(a, b), (a, c), (c, d)\} \rangle.$$

Then $\mathcal{B} \sqsubseteq \mathcal{A}$. By ignoring the indices, $\mathcal{A}$ is confluent, but $\mathcal{B}$ is not. The non-confluence of $\mathcal{B}$ follows from the non-confluence of $\rightarrow_\imath$ in $\mathcal{A}$.

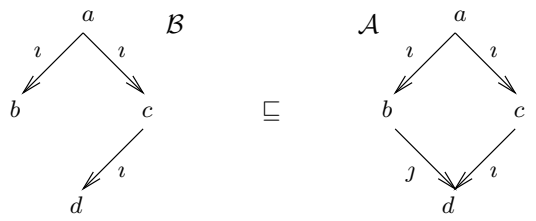


Figure 1.2: $\mathcal{B} \sqsubseteq \mathcal{A}$, $\mathcal{A}$ is confluent, $\mathcal{B}$ is not confluent

Remark 1.1.19 The notion of indexed sub-ARS is introduced here as a short-cut to some results that we will need. We introduce here the notion of indexed sub-ARS because in Chapters 4 and 5 we will consider such substructures of the context calculus λc:

- we will restrict the set of λc-terms by typing,
- we will consider a subset of rewrite relations of λc on the well-typed terms. Some rewrite relations of the context calculus λc do not apply to well-typed terms: these rewrite relations will be left out of the definitions. Also, in λc^{λ}, we will consider only the strong normalisation of $\rightarrow_c$, which excludes the β-rewrite relation.

The notion of indexed sub-ARS is a short-cut because by Lemmas 1.1.15 and 1.1.16 we could have used the notion of sub-ARS instead, and studied the rewrite relations and their interaction separately.

1.2 Lambda calculus

Lambda calculus is a rewrite system that formalises the notion of computable function. It is an example of a rewrite system with binders.

In this section we describe the untyped lambda calculus and a number of typed lambda calculi. In the untyped lambda calculus, expressions, called λ-terms, are freely formed, whereas in a typed lambda calculus the formation of λ-terms (and types) is guarded by a set of typing rules. There are two styles in typing λ-terms: typing by explicitly annotating variables with their type (also called typing à la Church) and typing by type assignment (also called typing à la Curry).

This section is organised as follows. In Section 1.2.1, the untyped version of lambda calculus is described. In Sections 1.2.2 and 1.2.3 examples of typed lambda calculi will be considered. In Section 1.2.2 the system $\lambda^{\rightarrow}$ is presented in both à la Church and à la Curry typing styles. Also, in this section, some terminology and notions will be introduced that are common to all typed lambda calculi which we consider here. In Section 1.2.3, Barendregt's lambda cube is presented, which is typed à la Church.

For more on lambda calculi, the reader is referred to [HS86] and [Bar84]; for more on typed lambda calculi, see [Bar92].

1.2.1 Untyped lambda calculus

We adopt for the most part the notions and notations as employed in [Bar84]. The results and their proofs can be found at the same reference.

Let $\mathcal{V}$ be a countably infinite set of variables. Typical elements of $\mathcal{V}$ will be denoted by $x, y, z, x', x_1, \ldots$

Definition 1.2.1 (λ-terms) The terms of λ-calculus, or λ-terms for short, are inductively defined by

$$M ::= x \mid (M\,M) \mid (\lambda x.\, M).$$

The set of λ-terms is denoted by Λ. A λ-term of the form $M\,M$ is called an application; a λ-term of the form $\lambda x.\, M$ is called an abstraction, where the symbol λ is called the abstractor.

Informally, we will also call λx an x-abstractor, and in an abstraction $\lambda x.\, M$ we will call M the body of the abstraction.

Notation. The outermost parentheses will be left out. Application associates to the left, so we write $M\,M_1 \ldots M_n$ instead of $((M\,M_1) \ldots M_n)$. Consecutive abstractions $(\lambda x_1. \ldots (\lambda x_n.\, M))$ will be abbreviated by $\lambda x_1, \ldots, x_n.\, M$. If no confusion can arise, λ-terms are called terms for short. If not explicitly stated otherwise, arbitrary terms will be denoted by $M, N, M_1 \ldots$.

Definition 1.2.2 (Subterms)

i) Let M be a λ-term. The set of subterms of M, denoted by $\textsc{Sub}(M)$, is defined inductively by

$$\begin{array}{lcl} \textsc{Sub}(x) & = & \{x\} \\ \textsc{Sub}(M_1\,M_2) & = & \textsc{Sub}(M_1) \cup \textsc{Sub}(M_2) \cup \{M_1\,M_2\} \\ \textsc{Sub}(\lambda x.\, M') & = & \textsc{Sub}(M') \cup \{\lambda x.\, M'\}. \end{array}$$

ii) Let M, N be two λ-terms. The term M is a subterm of N if $M \in \textsc{Sub}(N)$. If in addition $M \not\equiv N$, then M is called a proper subterm of N.

Definition 1.2.3 (Free variables in a λ-term) Let M be a λ-term.

i) The set of free variables of M, denoted by $\textsc{FVar}(M)$, is defined inductively by

$$\begin{array}{lcl} \textsc{FVar}(x) & = & \{x\} \\ \textsc{FVar}(M_1\,M_2) & = & \textsc{FVar}(M_1) \cup \textsc{FVar}(M_2) \\ \textsc{FVar}(\lambda x.\, M') & = & \textsc{FVar}(M') \setminus \{x\}. \end{array}$$

ii) The λ-term M is called closed if $\text{FVAR}(M) = \emptyset$.

A variable x may occur free or bound in a λ-term M. An occurrence of x in M is bound if it is in a subterm of M of the form $\lambda x.N$; otherwise it is free.

Definition 1.2.4 (Variable capturing) Variable capturing is a side-effect of a transformation by which a free variable occurrence becomes bound.

Definition 1.2.5 (λ-contexts)

i) A λ-context is a λ-term with some holes, denoted by $[]$, in it. That is, a λ-context is defined by

$$C ::= x \mid [] \mid CC \mid \lambda x.C.$$

If C has n holes, we also call it an n-context. If $C \equiv []$ then we call C trivial.

ii) Let C be an n-context and let $M_1, \ldots, M_n$ be n λ-terms where $n \geq 0$. Then $C[M_1, \ldots, M_n]$ denotes the result of filling the i^{th} hole by M_i, for $1 \leq i \leq n$. More precisely, if the occurrences of $[]$ in C are numbered from left to right, viz. $[]_1, \ldots, []_n$, then ($\vec{M}$ abbreviates $M_1, \ldots, M_n$)

$$\begin{array}{rcll} x[\vec{M}] & \equiv & x & \\ []_i[\vec{M}] & \equiv & M_i & \text{for } 1 \leq i \leq n \\ (C_1 C_2)[\vec{M}] & \equiv & (C_1[\vec{M}])(C_2[\vec{M}]) & \\ (\lambda x.C')[\vec{M}] & \equiv & \lambda x.C'[\vec{M}]. & \end{array}$$

This operation is called hole filling.

iii) Let C be an n-context and let $D_1, \ldots, D_n$ be n λ-contexts with $n \geq 0$ where D_i has k_i holes, for $1 \leq i \leq n$. Then $C[D_1, \ldots, D_n]$ denotes the result of filling the i^{th} hole by D_i, for $1 \leq i \leq n$. More precisely, if the occurrences of $[]$ in C are numbered from left to right, viz. $[]_1, \ldots, []_n$, then ($\vec{D}$ abbreviates $D_1, \ldots, D_n$)

$$\begin{array}{rcll} x[\vec{D}] & \equiv & x & \\ []_i[\vec{D}] & \equiv & D_i & \text{for } 1 \leq i \leq n \\ (C_1 C_2)[\vec{D}] & \equiv & (C_1[\vec{D}])(C_2[\vec{D}]) & \\ (\lambda x.C')[\vec{D}] & \equiv & \lambda x.C'[\vec{D}]. & \end{array}$$

This operation is called composition, and it results in a $\sum_{1 \leq i \leq n} k_i$-context.

Remark 1.2.6 Note that after the hole filling (and composition), variable capturing may occur: some free variables of $M_1, \ldots, M_n$ (or $D_1, \ldots, D_n$ respectively) may become bound by the binders of C. Note also that hole filling results in a term, while composition results in a context.

Notation. Arbitrary λ-contexts will be denoted by $C, D, C_1, \ldots$.

Terms of λ-calculus are considered to be equal up to renaming of bound variables. For example, the term $\lambda x.\, x\, z$ is considered to be equal to $\lambda y.\, y\, z$. Usually this renaming is defined stepwise and it is called α-reduction. The resulting equivalence is called α-conversion and it is often denoted by $=$, where the label α has been dropped. We do not give the definition of α-reduction here; see for example [Bar84].

Remark 1.2.7 This remark concerns the use of $\equiv$ (identity) and $=$ (α-conversion) between terms, (meta-)contexts, and later, types and (pseudo-)expressions in lambda calculi and context calculi in the remainder of this thesis.

Let a and b denote two terms, (meta-)contexts, types or (pseudo)-expressions. In general, we will write $a \equiv b$ if the equality between a and b does not involve α-conversion. We will write $a = b$ if the equality between a and b may involve α-conversion, provided of course that α-conversion is defined for a and b.

More precisely, we use $\equiv$ and $=$ as follows.

i) If a and b are two terms or two (pseudo-)expressions, then $a \equiv b$ denotes the equality between a and b which does not involve α-conversion, and $a = b$ denotes the equality between a and b where α-conversion is allowed.

ii) If a and b are two (meta-)contexts, then we use only the equality $\equiv$, without α-conversion, because α-conversion is not defined on (meta-)contexts. We do so in order to stress the absence of α-conversion.

iii) If a and b are two types (in a certain calculus), then we prefer to use $=$ to denote the equality between a and b. If α-conversion is defined on types in the calculus in question, we use $\equiv$ only in the cases where the equality between a and b does not involve α-conversion.

Definition 1.2.8 (Substitution) Let M and $M_1, \ldots, M_n$ be $n+1$ λ-terms, and let $x_1, \ldots, x_n$ be n distinct variables, for $n \geq 0$. The result of substitution of $M_1, \ldots, M_n$ for (the free occurrences of) $x_1, \ldots, x_n$ in M, denoted by $M[\![x_1 := M_1, \ldots, x_n := M_n]\!]$, or by $M[\![\vec{x} := \vec{M}]\!]$ for short, is defined by induction to M as

$$\begin{array}{lcl} x[\![\vec{x} := \vec{M}]\!] & = & \left\{ \begin{array}{lcl} M_i & : & \text{if } x \equiv x_i \text{ for certain } i \text{ with } 1 \leq i \leq n \\ x & : & \text{otherwise} \end{array} \right. \\ (M_1\, M_2)[\![\vec{x} := \vec{M}]\!] & = & (M_1[\![\vec{x} := \vec{M}]\!])(M_2[\![\vec{x} := \vec{M}]\!]) \\ (\lambda x.\, M')[\![\vec{x} := \vec{M}]\!] & = & \lambda x.\, (M'[\![\vec{x} := \vec{M}]\!]). \end{array}$$

In the last case, in order to avoid unintended variable capturing, it is assumed that bound variables x of M are renamed before applying the substitution if necessary, that is, if $x \in \text{FVAR}(\vec{M})$.

Remark 1.2.9 Variable capturing can be intended, as in $(\lambda x.\, [])[x] \equiv \lambda x.\, x$, or unintended, as in $(\lambda x.\, y)[\![y := x]\!] \neq \lambda x.\, x$. Unintended variable capturing is also called confusion of bound variables (cf. Section 3D2 in [CF58]).

Variable convention. We assume bound variables are renamed whenever necessary to avoid unintended variable capturing.

Remark 1.2.10 Without loss of generality, we can assume that in λ-terms there are no 'overshadowed' binders, like the leftmost λx in $\lambda x.\, \lambda x.\, x$. In a λ-term, such overshadowed binders can be renamed into a fresh variable. In the example, the leftmost λx can be renamed into λy by α-conversion, so, $\lambda x.\, \lambda x.\, x =_\alpha \lambda y.\, \lambda x.\, x$.

The fact that 'overshadowed' binders occur in the lambda calculus with name-carrying notation can be seen as an imperfection of the notation. If one considers bound variables as pointers to the binders by which they are bound, then the way of avoiding 'overshadowed' binders as described above is natural.

An analogous renaming of 'overshadowed' binders in λ-contexts will be allowed in our applications of λ-contexts: an 'overshadowed' binder λx of a context C may be renamed as λy if and only if y does not occur free in the λ-terms or λ-contexts that will (eventually) be put into the holes of C.

However, if one also considers applications where 'overshadowing' plays a role, as for example in object-oriented programming languages, such renaming is not allowed. In object-oriented programming languages overshadowing is encountered as method overriding. An example of method overriding in the λ-calculus format is the following. Consider two contexts C and D, both containing only one hole, and a term P. Let x occur free in P and let the hole of C be in the scope of the binder λx. Here C represents a class in which the method x is defined, D its subclass and P a program using the method x. Let $M \equiv C[D[P]]$ be a closed term. If the hole of D is in the scope of the binder λx, then x of P is bound by that binder of D; we say the method x of C is overridden by the method x of D. Otherwise, x of P is bound by the binder λx of C.

Definition 1.2.11 (Rewriting relations) The lambda calculus is defined on λ-terms with rewrite relations induced by the following rewrite rule schemas.

$$\begin{array}{lll} (\lambda x.\, M)\, N \to M[\![x := N]\!] & & (\beta) \\ \lambda x.\, M\, x \to M & \text{if } x \notin \mathrm{FVAR}(M) & (\eta) \end{array}$$

Remark 1.2.12 When we say that 'a rewrite relation $\to_\iota$ is induced by a rewrite rule schema (ι) : $L \to R$', we mean that the relation $\to_\iota$ is the compatible closure of the scheme (ι) (cf. Definition 3.1.5. in [Bar84]). Alternatively, we could say the relation $\to_\iota$ is generated by (ι) : $C[L] \to C[R]$ where C is a context (see Lemma 3.1.9. in [Bar84]).

Remark 1.2.13 Sometimes, a restricted inverse relation of the one generated by the rewrite rule (η) is used. This relation is generated by the rewrite rule schema:

$$M \to \lambda x.\, M\, x \qquad (\overline{\eta})$$

where $\overline{\eta}$ is not allowed to create a β-redex and $x \notin \mathrm{FVAR}(M)$. The effect of this relation is that in a typed version, in $\overline{\eta}$-normal forms, all functional symbols (i.e.

functions and variables of a functional type) are provided with the right number of arguments. For example,

$$\begin{array}{ll} & \lambda f^{(\mathtt{a}\to\mathtt{a})\to(\mathtt{b}\to\mathtt{b})\to\mathtt{c}}.\, f\,(\lambda x^{\mathtt{a}}.\, x) \\ \to_{\overline{\eta}} & \lambda f^{(\mathtt{a}\to\mathtt{a})\to(\mathtt{b}\to\mathtt{b})\to\mathtt{c}}.\, \lambda y^{\mathtt{b}\to\mathtt{b}}.\, f\,(\lambda x^{\mathtt{a}}.\, x)\,y \\ \to_{\overline{\eta}} & \lambda f^{(\mathtt{a}\to\mathtt{a})\to(\mathtt{b}\to\mathtt{b})\to\mathtt{c}}.\, \lambda y^{\mathtt{b}\to\mathtt{b}}.\, f\,(\lambda x^{\mathtt{a}}.\, x)\,(\lambda z^{\mathtt{b}}.\, y\,z). \end{array}$$

Remark 1.2.14 Often we will consider only the rewrite rule (β) on λ-terms. If the rewrite rule (η) is included in a lambda calculus, we will explicitly denote it in the name of the calculus; for example λ_η.

Remark 1.2.15 The rewrite relations and substitution are defined only on the set of λ-terms; they are not defined on λ-contexts.

Remark 1.2.16 Lambda terms with the rule (β) implement an abbreviation mechanism in the following sense. Consider a redex $(\lambda x.\, M)\,N$. By the rule (β), this redex is equal to $M[\![x := N]\!]$. An alternative reading of this substitution is 'let $x := N$ in M'. In other words, x can be seen as an abbreviation for N in M. Such an abbreviation mechanism is especially profitable if N is big and if x occurs in M many times.

This is a rather primitive notion of an abbreviation mechanism, where all free occurrences of x in M are replaced at the same time by the β-step. For a more advanced notion, see [SP94]. There, lambda calculus (actually, PTSs) has been extended with a definition mechanism, where definition unfolding is explicit and it is performed per free occurrence of x in M.

We introduce some rewriting terminology and list some well-known results. The terminology builds upon the rewriting terminology as given for ARSs. The results and their proofs can be found in [Bar84].

Let $C[(\lambda x.\, M)\,N] \to_\beta C[M[\![x := N]\!]]$. Then the subterm $(\lambda x.\, M)\,N$ is called a β-redex, and $M[\![x := N]\!]$ is called its contractum. We say that $C[(\lambda x.\, M)\,N]$ has a redex, and that it β-reduces to $C[M[\![x := N]\!]]$; the term $C[M[\![x := N]\!]]$ is called its one-step β-reduct. If in the rewrite step the context C is a trivial one, then the rewrite step is called a contraction.

Lemma 1.2.17 (Substitution lemma) *Let M, N, P be λ-terms, and let x, y be variables such that $x \notin \mathrm{FVAR}(P)$ and $x \not\equiv y$. Then*

$$M\,[\![x := N]\!]\,[\![y := P]\!] = M\,[\![y := P]\!]\,[\![x := N[\![y := P]\!]]\!].$$

Theorem 1.2.18 *The rewriting generated by $\to_\beta$ is confluent.*

Theorem 1.2.19 *The rewriting generated by $\to_\eta$ is confluent.*

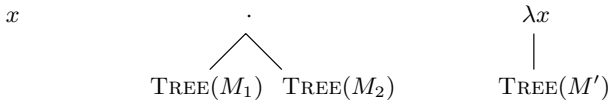
Lemma 1.2.20 *The rewrite relations $\to_\beta$ and $\to_\eta$ commute with each other.*

Theorem 1.2.21 *The rewriting generated by $\rightarrow_{\beta\eta}$ is confluent.*

Remark 1.2.22 The untyped lambda calculus is not WN, because $\rightarrow_\beta$ is not WN. Consider the λ-term $\Omega \equiv (\lambda x.\, x\, x)(\lambda x.\, x\, x)$. Then $\Omega \rightarrow_\beta \Omega$. Since Ω is its only β-reduct, this term has no β-normal form. Because the untyped lambda calculus is not WN, it is also not SN.

In this thesis we will consider λ-trees as an alternative notation for λ-terms.

Definition 1.2.23 (λ-trees) Let M be a λ-term. Then the corresponding λ-tree $\textsc{Tree}(M)$ is defined inductively as follows: the λ-trees of x, $M_1\, M_2$ and $\lambda x.\, M'$ are respectively,



Definition 1.2.24 (Root, leaf, path, spine) The notions of root, leaf and path are defined as usually on trees. The path from the root of a λ-tree to its leftmost leaf is called the spine of the λ-tree.

With each λ-term, a set of positions is associated. Positions are elements of $\{0,1\}^*$. The empty sequence over $\{0,1\}^*$ is denoted by ϵ. Two positions can be compared as strings, by the prefix ordering. The (proper) prefix ordering is denoted by $\prec$ and its reflexive closure by $\preceq$.

Definition 1.2.25 (Positions of a λ-term) Let M be a λ-term. The set of positions in M, denoted by $\textsc{Pos}(M)$, is defined as

$$\begin{array}{lcl} \textsc{Pos}(x) & = & \{\epsilon\} \\ \textsc{Pos}(M_1\, M_2) & = & \{\epsilon\} \cup \{0\varphi \mid \varphi \in \textsc{Pos}(M_1)\} \cup \{1\varphi \mid \varphi \in \textsc{Pos}(M_2)\} \\ \textsc{Pos}(\lambda x.\, M') & = & \{\epsilon\} \cup \{0\varphi \mid \varphi \in \textsc{Pos}(M')\}. \end{array}$$

A position in a λ-term M uniquely defines a subterm of M.

Definition 1.2.26 Let M be a λ-term, and let $\varphi \in \textsc{Pos}(M)$. Then $M|_\varphi$ is defined as

$$\begin{array}{lcl} M|_\epsilon & \equiv & M \\ (M_0\, M_1)|_{0\varphi} & \equiv & M_0|_\varphi \\ (M_0\, M_1)|_{1\varphi} & \equiv & M_1|_\varphi \\ (\lambda x.\, M)|_{0\varphi} & \equiv & M|_\varphi. \end{array}$$

Definition 1.2.27 (Descendants of positions) Let $M \rightarrow_\beta N$ be a β-step in which the redex occurrence at the position φ is contracted. Let $\psi \in \textsc{Pos}(M)$. The descendants of ψ in N over the rewrite step $M \rightarrow_\beta N$, denoted by $\textsc{Des}(M, \varphi, \psi)$, are defined as

- if $\varphi \not\preceq \psi$ then $\textsc{Des}(M, \varphi, \psi) = \{\psi\}$;
- if $\psi = \varphi 00\psi'$ then $\textsc{Des}(M, \varphi, \psi) = \{\varphi\psi'\}$;
- if $\psi = \varphi 1\psi'$ then $\textsc{Des}(M, \varphi, \psi) = \{\varphi\varphi'\psi' \mid \varphi' \in X\}$ where $X = \{\varphi' \in \textsc{Pos}(M) \mid M|_{\varphi'} \equiv x\}$
- otherwise, $\textsc{Des}(M, \varphi, \psi) = \emptyset$.

Descendants of a subterm P in M are traced by the position of P in M, that is, if $P \equiv M|_\psi$ and if $M \to_\beta N$ be a β-step in which the redex occurrence at the position φ is contracted, then the descendants of P are the terms $N|_{\psi'}$ where $\psi' \in \textsc{Des}(M, \varphi, \psi)$.

In an extension of the untyped lambda calculus or a typed lambda calculus, the property of preservation of strong normalisation is often considered, in addition to normalisation properties of Definition 1.1.6. We formulate here this property.

Definition 1.2.28 Let λ^* be an extension of (the untyped or a typed) lambda calculus. The rewriting of λ^* has the property of preservation of strong normalisation if λ-terms which are strongly normalising with respect to the rewriting of the lambda calculus are strongly normalising with respect to the rewriting of λ^*.

Remark 1.2.29 Sometimes a set $\mathcal{C}$ of variables is fixed. The elements of $\mathcal{C}$ are then considered to be bound externally; that is, they are not considered free, they are not bound nor substituted in a λ-term, and, in a typed system, they have a fixed type. The elements of $\mathcal{C}$ are called constants or function symbols, and the set $\mathcal{C}$ is also called an alphabet or signature.

1.2.2 The simply typed lambda calculus $\lambda^{\to}$

In this section the simply typed lambda calculus $\lambda^{\to}$ is presented in two versions, à la Church and à la Curry. The section starts with definitions that are common to both versions. Moreover, this section introduces some notions, notations and terminology which are common to all typed lambda calculi that we consider.

Definition 1.2.30 (Types) The simple types τ are defined over the set $\mathcal{V}_T$ of type variables using the function constructor $\to$, that is, if $\mathtt{a} \in \mathcal{V}_T$, then

$$\tau ::= \mathtt{a} \mid \tau \to \tau,$$

where $\to$ associates to the right. The set of types is denoted by $\textsc{Typ}(\lambda^{\to})$.

Remark 1.2.31 Type variables are often called base types.

Definition 1.2.32 (Type substitution) Let $\mathtt{a} \in \mathcal{V}_T$ and let τ and σ be types. Then the result of substituting τ for $\mathtt{a}$ in σ, denoted by $\sigma[\![\mathtt{a} := \tau]\!]$, is defined inductively:

$$\begin{array}{rcl} \mathtt{b}[\![\mathtt{a} := \tau]\!] & = & \begin{cases} \tau & \text{if } \mathtt{a} = \mathtt{b} \\ \mathtt{b} & \text{otherwise} \end{cases} \\ (\sigma_1 \to \sigma_2)[\![\mathtt{a} := \tau]\!] & = & (\sigma_1[\![\mathtt{a} := \tau]\!]) \to (\sigma_2[\![\mathtt{a} := \tau]\!]). \end{array}$$

The (pseudo-)terms of typed lambda calculi are λ-terms, or λ-terms where the variable in an abstraction is annotated with its type. (Term) substitution is defined as in the untyped λ-calculus; we will not give the definition here. A (pseudo-)term is called well-typed (with respect to a particular typed lambda calculus) if it can be typed by the typing rules. Let $\textsc{Ter}(\lambda^{\to})$ denote the set of well-typed λ-terms.

Both systems $\lambda^{\to}$ à la Curry and à la Church deal with statements, declarations and bases. A *statement* is of the form $M : \sigma$ with $M \in \Lambda$, and $\sigma \in \textsc{Typ}(\lambda^{\to})$. A *declaration* is a statement of the form $x : \sigma$ where $x \in \mathcal{V}$. A *basis* is a set of declarations with distinct variables $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. If $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ then the *domain of* Γ, denoted by $dom(\Gamma)$ is $\{x_1, \ldots, x_n\}$.

Remark 1.2.33 Bases are also called contexts. We avoid this terminology for obvious reasons.

Notation. We will often write $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ instead of $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. Also, we will write $\Gamma, x : \sigma$ for $\Gamma \cup \{x : \sigma\}$.

Definition 1.2.34 (Basis substitution) Let Γ be a basis, and let $[\![\mathtt{a} := \tau]\!]$ be a type substitution. Then

$$\Gamma[\![\mathtt{a} := \tau]\!] \;=\; \{(x :\; \sigma[\![\mathtt{a} := \tau]\!]) \mid (x : \sigma) \in \Gamma\}.$$

In both systems, the main rewrite relation is generated by the rewrite schema (β). If also the rewrite schema (η) or $(\overline{\eta})$ is used, it is explicitly noted, like in $\lambda^{\to}_{\eta}$ or $\lambda^{\to}_{\overline{\eta}}$ (see also Remark 1.2.14).

The system $\lambda^{\to}$-Church

The terms of $\lambda^{\to}$-Church are λ-terms decorated with types.

Definition 1.2.35 (Terms of $\lambda^{\to}$-Church) The terms of $\lambda^{\to}$-Church are inductively defined by

$$M ::= \; x \;\mid\; (M\,M) \;\mid\; (\lambda x^{\tau}.\, M).$$

Definition 1.2.36 (Typing in $\lambda^{\to}$ à la Church) A term $M \in \Lambda$ is typable by τ from the basis Γ, denoted by $\Gamma \vdash M : \tau$, if $\Gamma \vdash M : \tau$ can be derived using the typing rules displayed in Figure 1.3.

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma \vdash x:\tau}$$

$$(abs)\quad \frac{\Gamma, x:\tau \vdash M:\tau'}{\Gamma \vdash (\lambda x^{\tau}.M):\tau\to\tau'}$$

$$(app)\quad \frac{\Gamma \vdash M:\tau\to\tau' \qquad \Gamma \vdash N:\tau}{\Gamma \vdash MN:\tau'}$$

Figure 1.3: Typing rules for $\lambda^{\to}$ à la Church

Remark 1.2.37 The notion of derivation is here as described in [Bar92] on pages 36–38. The same notion of derivation, with different sets of typing rules, will be employed in other typed lambda calculi.

Remark 1.2.38 In fact, the 'rule' (var) is an axiom, because its assumption, that is, $(x:\tau)\in\Gamma$ does not involve a derivation. We present it as a rule and refer to all (var), (abs) and (app) as rules, for brevity. We will do the same in $\lambda^{\to}$ à la Curry and later, in the typing systems of the context calculus.

If $\Gamma \vdash M:\tau$ then we say Γ yields M in τ. Also, if τ is a type and if there are a basis Γ and a term M such that $\Gamma \vdash M:\tau$, then we say that τ is inhabited (by M).

Remark 1.2.39 If one considers also $\lambda^{\to}$-Church with function symbols $\mathcal{C}$, then all $f\in\mathcal{C}$ are considered to have a fixed type, and are typed by the rule (var). Alternatively, one could add a new rule

$$(const)\quad \frac{(f:\tau)\in\mathcal{C}}{\Gamma \vdash f:\tau}$$

to type the function symbols.

Contexts in $\lambda^{\to}$ are defined in an analogous way as in the untyped λ-calculus, except that the holes are annotated with their types and replacement of holes preserves types of holes.

Definition 1.2.40 (Contexts in $\lambda^{\to}$-Church)
Let $[]^{\tau}, []^{\sigma}, \ldots$ for $\tau,\sigma\in \textsc{Typ}(\lambda^{\to})$ denote typed holes.

i) A context is a well-typed term with some holes in it. In the typing rules, a hole $[]^{\tau}$ is treated as a free variable of type τ.

ii) Let $\Gamma, []^{\tau_1} : \tau_1, \ldots, []^{\tau_n} : \tau_n \vdash C : \tau$ be a context with n holes. Let $\Gamma \vdash M_i : \tau_i$ for $1 \leq i \leq n$. Then the term $C[M_1, \ldots, M_n]$ denotes the result of filling the i^{th} hole by M_i, for $1 \leq i \leq n$. This operation is called hole filling.

iii) Let $\Gamma, []^{\tau_1} : \tau_1, \ldots, []^{\tau_n} : \tau_n \vdash C : \tau$ be a context with n holes. Let $\Gamma \vdash D_i : \tau_i$ let k_i be the number of holes in D_i, for $1 \leq i \leq n$. Then $C[D_1, \ldots, D_n]$ denotes the result of filling the i^{th} hole by D_i, for $1 \leq i \leq n$. This operation is called composition, and it results in a context with $\sum_{1 \leq i \leq n} k_i$ holes.

Remark 1.2.41 The operations of hole filling and composition are well-defined because replacement of holes preserves typing, that is, if $\Gamma, []^{\tau_1} : \tau_1, \ldots, []^{\tau_n} : \tau_n \vdash C : \tau$ and $\Gamma \vdash M_i : \tau_i$ for $1 \leq i \leq n$ then $\Gamma \vdash C[M_1, \ldots, M_n] : \tau$. This can be shown analogously to the Substitution lemma below.

Some results in $\lambda^{\rightarrow}$ à la Church are listed.

Lemma 1.2.42 (Generation lemma)

i) *If* $\Gamma \vdash x : \sigma$ *then* $(x : \sigma) \in \Gamma$.

ii) *If* $\Gamma \vdash MN : \tau$ *then there is* σ *such that* $\Gamma \vdash M : \sigma \to \tau$ *and* $\Gamma \vdash N : \sigma$.

iii) *If* $\Gamma \vdash (\lambda x^{\sigma_1}.M) : \tau$ *then there is* σ_2 *such that* $\Gamma, x : \sigma_1 \vdash M : \sigma_2$ *and* $\tau = \sigma_1 \to \sigma_2$.

The set of terms of $\lambda^{\rightarrow}$ à la Church is closed under substitution for type variables and under substitution for term variables: this is stated by the next lemma.

Lemma 1.2.43 (Substitution lemma)

i) *If* $\Gamma \vdash M : \tau$ *then* $\Gamma [\![\mathtt{a} := \sigma]\!] \vdash M[\![\mathtt{a} := \sigma]\!] : \tau [\![\mathtt{a} := \sigma]\!]$.

ii) *Suppose* $\Gamma, x : \sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$. *Then* $\Gamma \vdash M[\![x := N]\!] : \tau$.

By the next lemma, the set of well-typed terms is closed under β-rewriting.

Lemma 1.2.44 (Subject reduction lemma) *Suppose* $M \twoheadrightarrow_\beta M'$. *Then, if* $\Gamma \vdash M : \tau$ *then* $\Gamma \vdash M' : \tau$.

By the next lemma, the set of well-typed terms is closed under η-rewriting and $\overline{\eta}$-rewriting, or, in other words, the set of well-typed terms is closed under η-conversion.

Lemma 1.2.45 (Subject reduction lemma for η and $\overline{\eta}$) *Suppose* $M \twoheadrightarrow_\eta M'$ *or* $M \twoheadrightarrow_{\overline{\eta}} M'$. *Then, if* $\Gamma \vdash M : \tau$ *then* $\Gamma \vdash M' : \tau$.

The well-typed terms are uniquely typed, and the typing is preserved under β-conversion.

Lemma 1.2.46

i) Suppose $\Gamma \vdash M : \tau$ *and* $\Gamma \vdash M : \tau'$. *Then* $\tau = \tau'$.

ii) Suppose $\Gamma \vdash M : \tau$ *and* $\Gamma \vdash M' : \tau'$ *and* $M =_\beta M'$. *Then* $\tau = \tau'$.

Theorem 1.2.47 *The rewriting generated by* $\rightarrow_\beta$ *is confluent.*

Theorem 1.2.48 *The rewriting generated by* $\rightarrow_{\beta\eta}$ *is confluent.*

Theorem 1.2.49 *The rewriting generated by* $\rightarrow_{\beta\overline{\eta}}$ *is confluent.*

Theorem 1.2.50 *The rewriting generated by* $\rightarrow_\beta$ *is strongly normalising.*

Proof: The proof is originally due to W.W. Tait [Tai67] (see also, for example Appendix 2 in [HS86]). The proof proceeds via a computability predicate on terms. QED

Lemma 1.2.51 *The rewriting generated by* $\rightarrow_{\overline{\eta}}$ *is strongly normalising.*

The set of $\overline{\eta}$-normal forms is closed under substitution and β-reduction.

Lemma 1.2.52

i) Let M, N *be* λ*-terms in* $\overline{\eta}$*-normal form, and let* x *be a variable. Let* x *and* N *be of the same type. Then* $M[\![x := N]\!]$ *is again in* $\overline{\eta}$*-normal form.*

ii) The rewriting generated by $\rightarrow_\beta$ *preserves* $\overline{\eta}$*-normal forms.*

The system $\lambda^\rightarrow$-Curry

The terms of $\lambda^\rightarrow$-Curry are the λ-terms as defined by Definition 1.2.1.

Definition 1.2.53 (Typing in $\lambda^\rightarrow$ à la Curry) A term $M \in \Lambda$ is typable by τ from the basis Γ, denoted by $\Gamma \vdash M : \tau$, if $\Gamma \vdash M : \tau$ can be derived using the typing rules displayed in Figure 1.4.

Lemma 1.2.54 (Generation lemma)

i) If $\Gamma \vdash x : \sigma$ *then* $(x : \sigma) \in \Gamma$.

ii) If $\Gamma \vdash MN : \tau$ *then there is* σ *such that* $\Gamma \vdash M : \sigma \rightarrow \tau$ *and* $\Gamma \vdash N : \sigma$.

iii) If $\Gamma \vdash (\lambda x.M) : \tau$ *then there are* σ_1, σ_2 *such that* $\Gamma, x : \sigma_1 \vdash M : \sigma_2$ *and* $\tau = \sigma_1 \rightarrow \sigma_2$.

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma \vdash x:\tau}$$

$$(abs)\quad \frac{\Gamma, x:\tau \vdash M:\tau'}{\Gamma \vdash (\lambda x.M):\tau\to\tau'}$$

$$(app)\quad \frac{\Gamma \vdash M:\tau\to\tau' \qquad \Gamma \vdash N:\tau}{\Gamma \vdash MN:\tau'}$$

Figure 1.4: Typing rules for $\lambda^{\to}$ à la Curry

The set of terms of $\lambda^{\to}$ à la Curry is closed under substitution and β-rewriting. The closure under substitution regards to both substitution for type variables and term variables. These properties are stated in the next two lemmas.

Lemma 1.2.55 (Substitution lemma)

i) If $\Gamma \vdash M:\tau$ *then* $\Gamma[\![\mathtt{a}:=\sigma]\!] \vdash M:\tau[\![\mathtt{a}:=\sigma]\!]$.

ii) Suppose $\Gamma, x:\sigma \vdash M:\tau$ *and* $\Gamma \vdash N:\sigma$. *Then* $\Gamma \vdash M[\![x:=N]\!]:\tau$.

Lemma 1.2.56 (Subject reduction lemma) *Suppose* $M \twoheadrightarrow_\beta M'$. *Then, if* $\Gamma \vdash M:\tau$ *then* $\Gamma \vdash M':\tau$.

The rewriting generated by $\to_\beta$ is complete in $\lambda^{\to}$ à la Curry.

Theorem 1.2.57 *The rewriting generated by* $\to_\beta$ *is confluent.*

Theorem 1.2.58 *The rewriting generated by* $\to_\beta$ *is strongly normalising.*

1.2.3 The lambda cube

The lambda cube, or the λ-cube for short, presents eight systems of typed lambda calculi in a uniform way. The uniform presentation has been generalised in Pure Type Systems (PTSs), which we will not study here. The systems of the λ-cube are typed à la Church.

For more information on the λ-cube, see [Bar92]; PTSs are described in for example [Ber88, Ter89, Bar92]. For the most part, our notions and notations agree with [Bar92]. See also Section 6.1.

The systems of the λ-cube are parametrised by the sorts of dependencies that may occur between terms and types. In order to control the dependencies, the set $\mathcal{S} = \{*, \Box\}$ of sorts is introduced. The dependencies between terms and types are then denoted by pairs over $\mathcal{S}$. A pair (s_1, s_2) denotes the dependency of the

expressions of sort s_2 on the expressions of sort s_1. Hence, each system of the λ-cube is parametrised by a set $\mathcal{R}$ of pairs over $\mathcal{S}$, which denote the dependencies which are allowed in that particular system.

Because terms and types may depend on each other, they are defined simultaneously as one syntactic category, called pseudo-expressions.

Definition 1.2.59 (Pseudo-expressions) The pseudo-expressions of the λ-cube are defined by

$$A ::= x \mid s \mid (\Pi x : A.\, A) \mid (\lambda x : A.\, A) \mid (A\, A)$$

where $s \in \mathcal{S}$.

Notation. As usual, standard abbreviations regarding brackets apply, including the association to the left in a sequence of applications $((A B_1) \ldots) B_n$. Consecutive abstractions $\Pi x_1 : A_1. \ldots \Pi x_n : A_n.\, B$ and $\lambda x_1 : A_1. \ldots \lambda x_n : A_n.\, B$ are abbreviated by $\Pi \vec{x} : \vec{A}.\, B$ and $\lambda \vec{x} : \vec{A}.\, B$, respectively. Furthermore, if $x \notin \text{FVAR}(B)$, we write $A \to B$ instead of $\Pi x : A.\, B$. Note that the variables in abstractions are annotated by their types in a different way than in $\lambda^{\to}$-Church: $\lambda x : A.\, B$ and $\Pi x : A.\, B$ in the λ-cube, versus $\lambda x^{\tau}.\, M$ in $\lambda^{\to}$-Church. However, we will sometimes also in the λ-cube write the type as a superscript, for the sake of readability.

As usual, we consider the pseudo-expressions to be equal up to the α-conversion. The notions of subterm and free variables are defined analogously to the same notion in the untyped lambda calculus. The notion of meta-contexts is not defined in the λ-cube.

Definition 1.2.60 (Substitution) Let A and $\vec{B}$ be $n+1$ pseudo-expressions and let $\vec{x}$ be n distinct variables. The result $A[\![\vec{x} := \vec{B}]\!]$ of substituting B_i for the free occurrences of x_i in A $(1 \leq i \leq n)$ is defined by induction to A as

$$\begin{array}{lcl}
x[\![\vec{x} := \vec{B}]\!] & = & \begin{cases} B_i & : \text{ if } x \equiv x_i \text{ for certain } i \text{ with } 1 \leq i \leq n \\ x & : \text{ otherwise} \end{cases} \\
s[\![\vec{x} := \vec{B}]\!] & = & s \\
(\Pi x : A_1.\, A_2)[\![\vec{x} := \vec{B}]\!] & = & \Pi x : (A_1[\![\vec{x} := \vec{B}]\!]).\, (A_2[\![\vec{x} := \vec{B}]\!]) \\
(\lambda x : A_1.\, A_2)[\![\vec{x} := \vec{B}]\!] & = & \lambda x : (A_1[\![\vec{x} := \vec{B}]\!]).\, (A_2[\![\vec{x} := \vec{B}]\!]) \\
(A_1\, A_2)[\![\vec{x} := \vec{B}]\!] & = & (A_1[\![\vec{x} := \vec{B}]\!])(A_2[\![\vec{x} := \vec{B}]\!]).
\end{array}$$

Definition 1.2.61 (Rewriting in the λ-cube) On the pseudo-expressions of the λ-cube, the β-rewrite relation is generated by

$$(\lambda x : A.\, B) C \to B\,[\![x := C]\!]\,. \qquad (\beta)$$

Definition 1.2.62 (The underlying calculus of the λ-cube) The calculus obtained from pseudo-expressions of the λ-cube equipped with the β-rewrite relation is called the underlying calculus of the λ-cube.

Lemma 1.2.63 *Rewriting in the underlying calculus of the λ-cube is confluent.*

Proof: The proof can be given via higher-order rewriting (see 1.3). One shows that the underlying calculus of the λ-cube is an orthogonal rewrite system. Then by Theorem 1.3.20, it is also confluent. QED

Lemma 1.2.64 (Substitution lemma for pseudo-expressions) *Let A,B and C be pseudo-expressions, let $x \notin \text{FVAR}(C)$ and let $x \neq y$. Then,*

$$A\,[\![x := B]\!]\,[\![y := C]\!] \;=\; A\,[\![y := C]\!]\,[\![x := B[\![y := C]\!]]\!].$$

Some terminology regarding typing is listed. A *statement* is of the form $A : B$ where A and B are pseudo-expressions. A *declaration* is a statement of the form $x : A$. A *pseudo-basis* is a finite ordered sequence of declarations with distinct variables. The empty basis is denoted by $\emptyset$. If $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ then the *domain of* Γ, denoted by $dom(\Gamma)$, is $\{x_1, \ldots, x_n\}$.

The typing rules of the λ-cube are given in Figure 1.5. The typing rules are divided into two groups:

i) the general axioms and rules, valid for all systems of the λ-cube.

ii) the specific rules, distinguishing between the eight systems; these are the Π-introduction rules parametrised by the allowed dependencies $\mathcal{R}$.

In the typing rules, $A, B, C, a, b, \ldots$ denote arbitrary pseudo-expressions, x, y, z, X, Y, ... denote arbitrary variables, and $s, s_1, s_2 \in \mathcal{S}$.

Definition 1.2.65 (Typing in the λ-cube) Let A and B be two pseudo-expressions and let Γ be a pseudo-basis. Then, $A : B$ can be derived from the pseudo-basis Γ, denoted by $\Gamma \vdash A : B$, if $\Gamma \vdash A : B$ can be derived using the typing rules displayed in Figure 1.5.

Definition 1.2.66 (The λ-cube) The eight systems of the λ-cube are displayed in Table 1.1. A system λS of the λ-cube is defined by taking the general rules and a subset of the specific rules parametrised by $\mathcal{R}$.

Notation. By $\Gamma \vdash A : B : C$ we denote $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$. Also, we will often say $A : B$ if there is Γ such that $\Gamma \vdash A : B$. If $x \notin \text{FVAR}(B)$ then $\Pi x : A.\, B$ is denoted as $A \to B$. Abstractions $\lambda X : *$ over type variables in elements of $\lambda 2$ are usually denoted by ΛX, like for example in the polymorphic version of the identity function $\Lambda X.\, \lambda x : X.\, x$. Note that the type of X (i.e. $*$) has been dropped. Moreover, in the type of such an element $\forall X$ is used instead of $\Pi X : *$, like for example in $(\Lambda X.\, \lambda x : X.\, x) \;:\; (\forall X.\, X \to X)$.

Some terminology regarding pseudo-bases and pseudo-expressions involved in typing is listed. A pseudo-basis Γ is called a *(legal) basis* if there are pseudo-expressions P and Q such that $\Gamma \vdash P : Q$. A pseudo-expression A is called a *(legal)*

General axiom and rules:

$$(axiom) \quad \emptyset \vdash * : \Box$$

$$(start) \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$(weak) \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad \text{if } x \notin \Gamma$$

$$(app) \quad \frac{\Gamma \vdash F : (\Pi x : A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : (B[\![x := A]\!])}$$

$$(abs) \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.\, B) : s}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi x : A.\, B)}$$

$$(conv) \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$$

Specific rules:

$$(Abs) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.\, B) : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{R}$$

Figure 1.5: Typing rules for the λ-cube

system	$\mathcal{R}$			
$\lambda^{\rightarrow}$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
λP	$(*,*)$		$(*,\Box)$	
$\lambda P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda P\omega = \lambda C$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

Table 1.1: The systems of the λ-cube

expression if there are a basis Γ and a pseuso-expression B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$. A pseudo-expression A is called a *type* if there are a basis Γ and $s \in \mathcal{S}$ such that $\Gamma \vdash A : s$. A pseudo-expression A is called an *element* if there are a basis Γ, a pseudo-expression B and $s \in \mathcal{S}$ such that $\Gamma \vdash A : B : s$.

The following results hold for each system of the λ-cube.

Lemma 1.2.67 (Free variable lemma) *Let $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ be a legal basis, say $\Gamma \vdash B : C$.*

i) The $x_1, \ldots, x_n$ are all distinct.

ii) $\text{FVAR}(B), \text{FVAR}(C) \subseteq \{x_1, \ldots, x_n\}$.

iii) $\text{FVAR}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ *for* $1 \leq i \leq n$.

Lemma 1.2.68 (Start lemma) *Let Γ be a legal basis.*

i) $\Gamma \vdash * : \Box$.

ii) If $(x : A) \in \Gamma$, then $\Gamma \vdash x : A$.

Lemma 1.2.69 *If $\Gamma \vdash A : B$, then either $B \equiv \Box$ or $\Gamma \vdash B : s$ for $s \in \mathcal{S}$.*

Lemma 1.2.70 (Thinning lemma) *Let Γ and Γ' be two legal bases. Suppose $\Gamma \subseteq \Gamma'$. Then, if $\Gamma \vdash A : B$ then $\Gamma' \vdash A : B$.*

Lemma 1.2.71 (Generation lemma)

i) If $\Gamma \vdash s : C$ then $s \equiv *$ and $C \equiv \Box$.

ii) If $\Gamma \vdash x : C$ then $\exists s \in \mathcal{S} \exists B$ such that $C =_\beta B$, $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$.

iii) If $\Gamma \vdash (\Pi x : A.\, B) : C$ then $\exists (s_1, s_2) \in \mathcal{R}$ such that $\Gamma \vdash A : s_1$, $\Gamma, x : A \vdash B : s_2$ and $C \equiv s_2$.

iv) *If* $\Gamma \vdash (\lambda x : A.b) : C$ *then* $\exists s \in \mathcal{S}\ \exists B$ *such that* $\Gamma \vdash (\Pi x : A.B) : s$, $\Gamma, x : A \vdash b : B$ *and* $C =_\beta (\Pi x : A.B)$.

v) *If* $\Gamma \vdash (F\,a) : C$ *then* $\exists A, B$ *such that* $\Gamma \vdash F : (\Pi x : A.B)$, $\Gamma \vdash a : A$ *and* $C =_\beta B[\![x := a]\!]$.

The set of legal expressions is closed under substitution.

Lemma 1.2.72 (Substitution lemma) *Suppose* $\Gamma, x : A, \Gamma' \vdash B : C$ *and* $\Gamma \vdash D : A$. *Then,* $\Gamma, \Gamma'[\![x := D]\!] \vdash B[\![x := D]\!] : C[\![x := D]\!]$.

By the next lemma, the set of legal expressions is closed under β-rewriting.

Lemma 1.2.73 (Subject reduction for the λ-cube) *If* $\Gamma \vdash A : B$ *and* $A \twoheadrightarrow_\beta A'$ *then* $\Gamma \vdash A' : B$.

Proof: A proof can be found in for example [Bar92]. QED

Theorem 1.2.74 (Confluence for the λ-cube) *The rewriting generated by* $\rightarrow_\beta$ *is confluent.*

Theorem 1.2.75 (Strong normalisation for the λ-cube) *All systems in the* λ*-cube are strongly normalising.*

Proof: A proof can be found in for example [GN91]. QED

Example 1.2.76 Some examples of legal expressions per level and a short analysis of their structure are given below. One can think of these typing statements as in λC.

- Kinds (i.e. elements of $\Box$): Some examples are listed below.

$$\begin{array}{lcll} \emptyset & \vdash & * & : \Box \\ X : * & \vdash & \Pi x : X.\, \Pi y : (\Pi z : X.*).* & : \Box \end{array}$$

  Thus, in general, kinds are of the form $\Pi \vec{x} : \vec{A}.*$ where $A_i : *$ or $A_i : \Box$ for $1 \leq i \leq |\vec{A}|$. Note that $*$ is a special element of $\Box$.

  Variables, λ-abstractions and applications are not a kind. In particular, that means also that there are no β-redexes such that $(\lambda x : A.B)C : \Box$.

  Both $*$ and $\Pi \vec{x} : \vec{A}.*$ may be inhabited.

- Types (i.e. elements of $*$): Some examples are listed below.

$$\begin{array}{lcll} X : * & \vdash & X & : * \\ \emptyset & \vdash & \Pi X : *.\, \Pi x : X.\, X & : * \\ \emptyset & \vdash & \Pi X : *.\, (\lambda Y : *.\, Y)\, X & : * \\ Y : * & \vdash & (\lambda X : *.\, \Pi x : X.\, X)\, Y & : * \\ F : (\Pi X : *.*), Y : * & \vdash & \Pi x : Y.\, F\, Y & : * \\ X : *, G : (\Pi x : X.*), y : X & \vdash & G\, y & : * \end{array}$$

Note that $*$ contains variables, Π-abstractions, and applications. Hence, $*$ contains redexes.

Types reduce to expressions of the form $\Pi\vec{x} : \vec{A}.\, B$ where $B : *$ and $B = x\vec{B}$ with x a variable.

Last but not least, types may be inhabited.

- Elements of kinds: Elements of kinds are called constructors in λC.

  A couple of examples are listed below.

$$\begin{array}{lcll}
F : (\Pi X : *.\, *) & \vdash & F & : \Pi X : *.\, * \\
X : *, G : (\Pi x : X.\, *) & \vdash & G & : \Pi x : X.\, * \\
X : * & \vdash & \lambda x : X.\, \Pi Y : *.\, Y & : \Pi x : X.\, * \\
\emptyset & \vdash & \lambda X : *.\, \lambda x : X.\, X & : \Pi X : *.\, \Pi x : X.\, * \\
Y : * & \vdash & (\lambda X : *.\, \lambda x : X.\, X)\, Y & : \Pi x : Y.\, *
\end{array}$$

  Elements of kinds may be variables, λ-abstractions and applications. Hence, such elements may also be β-redexes.

  Elements of kinds reduce to expressions of the form $\lambda\vec{x} : \vec{A}.\, \Pi\vec{y} : \vec{B}.\, C$ where $C : *$ and $C = z\vec{C}$ with z a variable.

  In general, elements of kinds have no inhabitants. However, elements of kinds which are provided with all arguments collapse into types. For example, the expression $(\lambda A : *.\, A)\, B$, which is an element of $*$, may be inhabited.

- Elements of types: Some examples are given below.

$$\begin{array}{lcll}
X : *, x : X & \vdash & x & : X \\
X : *, x : X, & & & \\
f : X \to X & \vdash & f\, x & : X \\
X : * & \vdash & \lambda x : X.\, x & : X \to X \\
\emptyset & \vdash & \lambda X : *.\, \lambda x : X.\, x & : \Pi X : *.\, X \to X \\
Y : * & \vdash & (\lambda X : *.\, \lambda x : X.\, x)\, (Y \to Y) & : (Y \to Y) \to (Y \to Y)
\end{array}$$

  Elements of types may be variables, λ-abstractions and applications. In elements of types, Π-abstractions may occur but only in the type T in a λ-abstraction $\lambda x : T.\, M$ or in an argument of an application $(\lambda X : *.\, \lambda x : X.\, x)\, T$. Elements of types may be redexes.

  Elements of types reduce to expressions of the form $\lambda\vec{x} : \vec{A}.\, y\vec{B}$ where y is a variable.

  Elements of types have no inhabitants.

Note that $\Box$ is not a subexpressions of any legal expression other than itself. That is, $\Box$ does not combine into a bigger legal expression. Also, one could prove that in a legal expression no variable is a place-holder for $\Box$ (trivial) or for elements of $\Box$ (because there are no legal variables x such that $x : \Box$).

1.3 Higher-order rewriting

Higher-order rewrite systems provide a framework for rewrite systems with binders. Although there is no generally accepted format of higher-order rewrite systems, the existing higher-order rewrite systems resemble each other in the following way. They are formed from two parts:

- a substitution calculus, which implements variable binding and substitution machinery in a uniform way; and
- an object-language consisting of a signature and rewrite rules, which implement a particular rewrite system and its computation.

The substitution calculus is specific to a class of higher-order rewrite systems. For example in Klop's Combinatory Reduction Systems ([Klo80]) the substitution calculus is the untyped lambda calculus with function symbols. The signature and rewrite rules are specific to a particular rewrite system represented in a particular higher-order format.

Different formats of higher-order rewrite systems have been studied in [Klo80, Wol93, Kha90]. In this thesis we use a concrete class of higher-order rewrite systems, namely the pattern rewrite systems (PRSs), which amount to Nipkow's Higher-Order Rewrite Systems (HRSs, [Nip91, Nip93, MN98], see also Section 3.5 in [Oos94]). The pattern rewrite systems are higher-order rewrite systems with $\lambda^{\rightarrow}_{\eta}$-Church with function symbols as the substitution calculus.

We adopt the notions and most of the notations as given in [Oos94, Oos95]. Because higher-order rewriting is not as widely known as for example abstract rewriting and lambda calculus, we will include more explanations and examples than in the preceding introductory sections. Furthermore, although PRSs can be considered as rewrite systems in their own right, we consider them as a framework for rewrite systems with binders. This helps us give more intuition about the hows and whys in higher-order rewriting. For more background on higher-order rewriting, the reader is referred to [Klo80, KOR93, MN98, Wol93, Oos94, Raa96].

Types, elements and meta-contexts

The substitution calculus of PRSs is $\lambda^{\rightarrow}_{\eta}$-Church with function symbols. In this section, we will refer to this calculus by $\lambda^{\rightarrow}_{\eta}$ for short. The types of $\lambda^{\rightarrow}_{\eta}$ have already been defined in the section about typed lambda calculi; we repeat it here.

Definition 1.3.1 The types of the substitution calculus are the types as defined in Definition 1.2.30.

In fact, because we are only interested in the applicability of an element of a PRS, we will often use the singleton $\{0\}$ for the set of type variables (or base types).

The function symbols of $\lambda^{\rightarrow}_{\eta}$ are related to the rewrite system which is represented in the higher-order format by a particular PRS. The function symbols are typed a priori. The set of function symbols, denoted by $\mathcal{C}$, is called a signature.

The elements of a PRS, called *preterms*, are the terms of the calculus $\lambda^{\rightarrow}_{\overline{\eta}}$-Church with function symbols of $\mathcal{C}$. This calculus has the same elements as the calculus $\lambda^{\rightarrow}$-Church with function symbols of $\mathcal{C}$.

Definition 1.3.2 Preterms are the well-typed $\lambda^{\rightarrow}$-terms, as defined by Definition 1.2.35 (and Remark 1.2.29) and typed by Definition 1.2.36.

Notation. Arbitrary preterms will be denoted by $s, t, \vec{s}, \ldots$. As usual, we consider preterms equal up to renaming of bound variables. Furthermore, we will often drop the type of the variable in an abstraction, for the sake of readability. In PRSs it is customary to drop the λ in abstractions. That is, we write $x.\, s$ instead of $\lambda x.\, s$. Consecutive abstractions $x.\, y.\, s$ are abbreviated by $x, y.\, s$.

Example 1.3.3 The running example in this section is the overused example of the representation $\mathcal{H}_\lambda$ of lambda calculus in the PRS-format. For this example, let the signature be the set $\mathcal{C}_\lambda = \{\mathsf{abs} : (0 \to 0) \to 0, \mathsf{app} : 0 \to 0 \to 0\}$. Examples of preterms of $\mathcal{H}_\lambda$ are

x
$\mathsf{abs}(x.\, x)$
$\mathsf{app}\,(\mathsf{abs}(x.\, x))\; y$
$z.\, \mathsf{abs}(x.\, zx)$
$\mathsf{app}\; z$ and
$\mathsf{abs}(x.\, (y.\, y)x)$.

In the definition of rewrite steps, meta-contexts play an important role. The meta-contexts used, which are here called *precontexts*, are the meta-contexts over preterms.

Definition 1.3.4 A precontext D is a preterm with some holes in it, as defined by Definition 1.2.40.

Consider now a rewrite system $\mathcal{A}$ and its representation in the PRS-format $\mathcal{H}_\mathcal{A}$. Among the preterms of $\mathcal{H}_\mathcal{A}$ are the representations of the elements of the rewrite system $\mathcal{A}$. The representation of the elements of $\mathcal{A}$ are typed by a base type; one may think of this type as the term type 0. These representations can be characterised as follows: a preterm s of $\mathcal{H}_\mathcal{A}$ is a representation of an element of $\mathcal{A}$ if and only if

- s starts with a function symbol, and all function symbols (including the starting symbol) in s have the right number of arguments (that is, the preterm s is a $\overline{\eta}$-normal form of a base type ('term type')),
- all variables of s are place-holders for other representations of the elements of $\mathcal{A}$ (that is, all variables of s are of a base type),

- in s, substitutions are computed, (that is, s is in β-normal form, because in a PRS it is the β-reduction that internalises substitution computation).

Example 1.3.5 In the example above, the 'meaningful' preterms are the variable x, which represents the variable x; the preterm $\mathsf{abs}(x.\,x)$, which represents the λ-term $\lambda x.\,x$; and the preterm $\mathsf{app}\,(\mathsf{abs}(x.\,x))\;y$, which represents the λ-term $(\lambda x.\,x)\,y$.

As witnessed by the definitions of rewrite relation (cf. Definitions 1.3.8 and 1.3.11), one cannot restrict attention only to such preterms. In particular, the rewrite relation in PRSs is defined via preterms l that in general are not of base type and may contain variables of types other than base types. However, one can (and does) focus attention only to preterms in $\beta\overline{\eta}$-normal form, with the benefit of having more grip on relevant preterms.

Definition 1.3.6 Terms are preterms in $\beta\overline{\eta}$-normal form.

Although no rewriting is allowed on meta-contexts, the notion of being in $\beta\overline{\eta}$-normal form can straightforwardly be extended to meta-contexts.

Definition 1.3.7 Contexts C are precontexts in $\beta\overline{\eta}$-normal form.

Rewrite relation in a PRS

Rewriting in a PRS is generated by a set of pattern rewrite rules. We will first define the rewrite relation(s) on terms, and then extend the definition to preterms.

Definition 1.3.8 A pattern is a term of the form $\vec{z}.\,f(\vec{s})$ such that

- $f(\vec{s})$ is of a base type, and
- each z_i among $\vec{z}$ occurs free in $f(\vec{s})$ and has only ($\overline{\eta}$-normal forms of) pairwise distinct variables not among $\vec{z}$ as arguments.

The function symbol f of the pattern $\vec{z}.\,f(\vec{s})$ is called the head symbol of the pattern.

Definition 1.3.9 A pattern rewrite rule $(\mathsf{i}) :\;\; l \to r$ is a pair of closed terms of the same type τ, where the left-hand side l is a pattern.

Example 1.3.10 An example of a pattern rewrite rule is

$$z_1, z_2.\,\mathsf{app}\,(\mathsf{abs}(x.\,z_1 x))\;z_2 \;\to\; z_1, z_2.\,z_1 z_2\;. \qquad (\mathsf{beta})$$

This rule represents the β-rewrite rule of λ-calculus.

Let s and t be terms. A rewrite step $s \to_{\mathsf{i}} t$ in a PRS consists of

- extracting the left-hand side $l \equiv \vec{z}.\,f(\vec{s})$ of a rewrite rule $(\mathsf{i}) :\;\; l \to r$ in s,

- literally replacing the left-hand side l by the right-hand side $r \equiv \vec{z}.\, r'$, and

- β-reducing to normal form.

By the extraction step an internalised substitution for $\vec{z}$ in $f(\vec{s})$ is formed, and by the β-reduction step the internalised substitution of variables $\vec{z}$ is applied to r'. This amounts to defining a rewrite step as it is done in Nipkow's HRSs: $C[(f(\vec{s}))^\sigma] \to_{\mathsf{i}} C[(r')^\sigma]$ where σ is a substitution ranging over the variables $\vec{z}$.

If a pattern l can be extracted from a term s, that is, if there is a context C such that $C[l] \twoheadrightarrow_\beta s$, then we call l a pattern (at C) in s. Note that along the rewrite sequence $C[l] \twoheadrightarrow_\beta s$, the left-hand side l is not duplicated, because C is in $\beta\overline{\eta}$-normal form. In particular, the head symbol of l has precisely one descendant in s. Hence, if (i) : $l \to r$ is a rewrite rule and l is a pattern at C in s, then l and C uniquely determine a redex occurrence in s.

Definition 1.3.11 Let s and t be two terms. Then $s \to_{\mathsf{i}} t$ if there is a rewrite rule (i) : $l \to r$ and a context C such that $s \twoheadleftarrow_\beta C[l]$ and $C[r] \twoheadrightarrow_\beta t$.

In fact, in PRSs, the context C of this definition is unique (cf. Proposition 3.2.17 in [Oos94]). This is due to the fact that C is in $\beta\overline{\eta}$-normal form.

Example 1.3.12 Consider the pattern rewrite rule (beta) given in the previous example. An example of a rewrite step is

$$\mathsf{app}\,(\mathsf{abs}(x.\,x))\; y \to_{\mathsf{beta}} y$$

because

$$\begin{array}{lll} \mathsf{app}\,(\mathsf{abs}(x.\,x))\; y & \twoheadleftarrow_\beta & (z_1, z_2.\,\mathsf{app}\,(\mathsf{abs}(x'.\,z_1 x'))\; z_2)(x.\,x)y \\ & \equiv & ([](x.\,x)y)[z_1, z_2.\,\mathsf{app}\,(\mathsf{abs}(x.\,z_1 x))\; z_2] \\ & \to_{repl} & ([](x.\,x)y)[z_1, z_2.\,z_1 z_2] \\ & \equiv & (z_1, z_2.\,z_1 z_2)(x.\,x)y \\ & \twoheadrightarrow_\beta & y. \end{array}$$

where $\to_{repl}$ denotes the replacement of l by r.

Definition 1.3.13 Let s and t be two preterms. Then, $s \to_{\mathsf{i}} t$ if and only if there are terms s', t', a rewrite rule (i) : $l \to r$ and a context C such that $s =_{\beta\overline{\eta}} s' \twoheadleftarrow_\beta C[l]$ and $C[r] \twoheadrightarrow_\beta t' =_{\beta\overline{\eta}} t$.

Pattern rewrite systems

Definition 1.3.14 A higher-order pattern rewrite system (PRS) is a pair $(\mathcal{C}, \mathcal{R})$ consisting of a signature $\mathcal{C}$ and a set $\mathcal{R}$ of pattern rewrite rules.

In general, the adjective *higher-order* in higher-order rewrite systems pertains to the order of variables in the rewrite rules.

Definition 1.3.15 The order function $\textsc{Ord}(\_)$ on types is defined by ($\mathsf{a} \in BT$)

$$\begin{array}{lcl} \textsc{Ord}(\mathsf{a}) & = & 1 \\ \textsc{Ord}(\vec{\tau} \to \mathsf{a}) & = & 1 + max\{\textsc{Ord}(\tau_i) \mid 1 \leq i \leq |\vec{\tau}|\}. \end{array}$$

The order of a rewrite rule (i) : $\vec{z}.\, f(\vec{s}) \to r$ is the maximal order of the types of the variables $\vec{z}$ and of the function symbols in it. The order of a PRS is the maximal order of its rewrite rules.

Example 1.3.16 The structure $\mathcal{H}_\lambda = (\mathcal{C}_\lambda, (\mathsf{beta}))$ is an example of a second-order PRS. The signature $\mathcal{C}_\lambda$ was defined in Example 1.3.3 and the rewrite rule (beta) in Example 1.3.10.

With each pattern rewrite system an (indexed) ARS can be associated in a straightforward way. The notions of (full) sub-ARS (Definition 1.1.13) and indexed sub-ARS (Definition 1.1.14) are then also straightforwardly carried over from abstract rewriting into PRSs. We will say a PRS $\mathcal{H}$ is a sub-PRS (or, an indexed sub-PRS) of $\mathcal{H}'$ if the underlying ARS of $\mathcal{H}$ is a sub-ARS (respectively, an indexed sub-ARS) of the underlying ARS of $\mathcal{H}'$.

Orthogonal pattern rewrite systems

Orthogonality describes the property of rewrite systems that contracting a redex in a term does not destroy other redexes that are present in the term. That is, in an orthogonal rewrite system the redexes in a term are, loosely speaking, independent of each other. Orthogonality implies confluence.

In the first-order rewriting, orthogonality is defined via properties of the left-hand sides of the rewrite rules. The notion of orthogonality can be lifted from first-order to higher-order rewriting.

Definition 1.3.17

i) A pattern $\vec{z}.\, f(\vec{s})$ is linear if each z_i occurs free exactly once in $f(\vec{s})$.

ii) A pattern rewrite rule (i) : $l \to r$ is left-linear if l is a linear pattern.

iii) A PRS is called left-linear if all its rewrite rules are left-linear.

Definition 1.3.18 A critical pair is a tuple $(C[r\vec{s}], r'\vec{t})$ where C is a context and $C[l\vec{s}] = l'\vec{t}$ is a most general overlapping between two redexes of rewrite rules (i) : $l \to r$ and (j) : $l' \to r'$. If the context is the trivial one, the rewrite rules must be different.

Definition 1.3.19 A pattern rewrite system is called orthogonal if there are no critical pairs and if all rewrite rules are left-linear.

Theorem 1.3.20 *Orthogonal PRSs are confluent.*

Proofs can be found in for example [Oos94, OR93].

Developments and descendants in linear PRSs

A development step is a simultaneous contraction of a set of redex occurrences in a term. A condition for performing a development step is that the redex occurrences involved are independent from each other. Development steps and this condition are formulated in the next two definitions.

Definition 1.3.21 Let C be a context with n holes. Let $(\mathrm{i}_1) : \quad l_1 \to r_1, \ldots$ $(\mathrm{i}_n) : \quad l_n \to r_n$ be n rewrite rules of a linear PRS. Let s and t be terms such that $s \twoheadleftarrow C[l_1, \ldots, l_n]$ and $C[r_1, \ldots, r_n] \twoheadrightarrow t$. Then we say that s rewrites to t in one development step and denote this by $s \multimap t$.

In the definition above, the terms s (and t) are specific terms, defined by the context C and the rewrite rules. A development step from an arbitrary s depends on the existence of patterns in s that can be extracted independently from each other.

Definition 1.3.22 A set $l_1, \ldots, l_n$ of patterns (at $C_1, \ldots, C_n$) in s is said to be independent if there is a context C with n holes such that $C[l_1, \ldots, l_n] \twoheadrightarrow_\beta s$, and that the head symbol of l_k descends to the same symbol in s in both $C[l_1, \ldots, l_n] \twoheadrightarrow_\beta s$, and $C_k[l_k] \twoheadrightarrow_\beta s$ for each $1 \leq k \leq n$.

As we have already mentioned, if l is a pattern at C in s, then l and C uniquely determine a redex occurrence in s. Accordingly, a set of redex occurrences is called independent if their patterns are independent.

In the definition above, the descendant relation along the rewrite sequences $C[l_1, \ldots, l_n] \twoheadrightarrow_\beta s$ and $C_k[l_k] \twoheadrightarrow_\beta s$ is, of course, the descendant relation defined for β-rewriting in untyped lambda calculus (cf. Definition 1.2.27). Furthermore, the first condition describes the independence of patterns in s, and the second condition requires that contexts C and C_k consider the same redex occurrence of l_k in s.

Example 1.3.23 The second condition prohibits 'cheating'. In this example we consider an another PRS than the running example, because the running example behaves too nicely.

Consider the pattern rewrite rules

$$z.\, f(g(z)) \to z.\, z \qquad (1)$$
$$z.\, g(z) \to z.\, z \qquad (2)$$

and let $s \equiv h(\underline{f(\overline{g(a)})}, g(b))$, where two redex occurrences are marked. Then, informally speaking, the two redex occurrences are not independent, because the contraction of the underlined redex destroys the overlined redex. That is, there is no context C that satisfies the conditions of the definition above.

Note though, that there is a context such that $C[z.\, f(g(z)), z.\, g(z)] \twoheadrightarrow_\beta s$, namely $C \equiv h([]a, []b)$. In this context, the other redex occurrence of the rule (2) is extracted.

Example 1.3.24 In the term

$$s \equiv \underline{\mathsf{app}\ (\mathsf{abs}(x.\ \overline{\mathsf{app}\ (\mathsf{abs}(y.\ y))\ x})\ z}$$

the underlined and overlined redex occurrences are independent. By contracting these redex occurrences, s rewrites to z in one development step, that is,

$$s \equiv \underline{\mathsf{app}\ (\mathsf{abs}(x.\ \overline{\mathsf{app}\ (\mathsf{abs}(y.\ y))\ x})\ z} \multimap z.$$

Lemma 1.3.25 *In an orthogonal PRS, every set of redex occurrences in a term s is independent.*

The descendants of a redex occurrence over a rewrite sequence are traced via the head symbol of the pattern related to the redex occurrence. Tracing descendants in PRSs is more difficult than in the first-order case, because the descendants of disjoint subterms may become nested after a rewrite step. Furthermore, the definition of a rewrite step involves β-expansion viz. $s \twoheadleftarrow_\beta C[l]$. Hence, the descendant relation in PRSs will involve tracing origins of function symbols of s in $C[l]$ along this expansion.

The descendant relation in PRSs is defined over a development step. Because each rewrite step is a development step of a singleton, the definition of descendant relation applies to rewrite steps too.

The descendant relation is defined in linear pattern rewrite systems. The property of linear PRSs that is used in the definition of the descendant relation is that in a rewrite sequence to a normal form $C[l_1, \ldots, l_n] \twoheadrightarrow_\beta s$, each function symbol of $C[l_1, \ldots, l_n]$ has precisely one descendant in s. That is, each function symbol of s has a (unique) origin in $C[l_1, \ldots, l_n]$.

The following definition and results are found in [Oos95]. We do not go into all the details of the definition. In particular, this definition employs tracing origins of function symbols of s over $s \twoheadleftarrow_\beta C[l_1, \ldots, l_n]$. For a precise definition of the descendant relation over a β-expansion, see Definition 3.1.25 in [Oos94].

Definition 1.3.26 The descendant relation of a development step $s \multimap t$ where $s \twoheadleftarrow_\beta C[l_1, \ldots, l_n]$ and $C[r_1, \ldots, r_n] \twoheadrightarrow_\beta t$, of a set of independent redexes is the relation composition of the descendant relations of its three components: the β-expansion, the replacement and the β-reduction. In the replacement step from $C[l_1, \ldots, l_n]$ to $C[r_1, \ldots, r_n]$ function symbols in the context C descend to themselves. Function symbols in the left-hand sides are said to be destroyed. Function symbols in the right-hand sides are said to be created.

Example 1.3.27 We consider a rewrite step and the descendants of the underlined and overlined redex occurrences:

$$\begin{array}{rcl} s & \equiv & \mathsf{app}\ (\mathsf{abs}(x.\ \underline{\mathsf{app}\ (\mathsf{abs}(y.\ x))\ y'}))\ (\overline{\mathsf{app}\ (\mathsf{abs}(z.\ z))\ z'}) \\ & \to_{\mathsf{beta}} & \underline{\mathsf{app}\ (\mathsf{abs}(y.\ (\overline{\mathsf{app}\ (\mathsf{abs}(z.\ z))\ z'}))\ y'} \\ & \equiv & t. \end{array}$$

Note that the two redex occurrences were disjoint in s and are nested in t. However, in both terms the redex occurrences are independent.

Lemma 1.3.28 *In an orthogonal PRS, the descendants of independent patterns are again independent.*

Lemma 1.3.29 *In an orthogonal PRS, the descendants of redexes are again redexes of the same rewrite rules.*

If the descendants of a redex are again redexes, they are called residuals.

1.4 General notation and conventions

We assume bound variables are renamed whenever necessary. For example, before doing a β-step from the λ-term $(\lambda x.\, \lambda y.\, x)y$, it is necessary to rename the bound variable y; otherwise, an unintended binding would occur. Also in a type derivation of the same λ-term the same bound variable y has to be renamed; otherwise the basis $\{y : \sigma,\ x : \tau,\ y : \rho\}$, which occurs along the derivation, would contain two declarations of the same variable. We understand the renaming of bound variables as a repairment of the short-comings of the notation with name-carrying variables.

If $\mathcal{A}$ is a calculus, then the set of terms of $\mathcal{A}$ will be denoted by $\textsc{Ter}(\mathcal{A})$, unless explicitly stated otherwise. Moreover, if $\mathcal{A}$ is a typed calculus, then the set of types of $\mathcal{A}$ will be denoted by $\textsc{Typ}(\mathcal{A})$.

The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

A finite sequence of elements, possibly alternated by a constructor will be abbreviated by a vector. For example, $\vec{a} = a_1, \ldots, a_n$, $\vec{\tau} \to \tau = \tau_1 \to (\tau_2 \to \ldots \to (\tau_n \to \tau))$ and $\vec{0} \to 0 = 0 \to (0 \to \ldots (0 \to 0))$. The empty sequence will be denoted by ϵ. The length of the vector $\vec{a}$ will be denoted by $|\vec{a}|$.

The identity relation between notions other than terms, types, contexts and (pseudo-)expressions, will be denoted by $=$ (see also Definition 1.1.2 and Remark 1.2.7).

Chapter 2

Contexts in lambda calculus

Contexts play a role in many systems of expressions and expression transformation, like for example in lambda calculus, in programming languages, and in linguistics. Different applications of contexts in these systems have led to different notions of context. Although the notions of context differ, the transformations involving contexts in these systems are common. Consequently, the problems that are encountered when formalising contexts and context-related transformations are common too.

In the untyped and simply typed lambda calculus, a context is a term with some holes in it. The distinctive feature of contexts is that when a term is filled into a hole of a context, some free variables of the term may become bound by the binders of the context. This feature is called variable capturing. The main problem in a context formalisation is that, when β-reduction is naïvely defined on contexts, the intended variable capturing may be lost. That is, β-reduction and filling of holes do not commute and consequently, the formalisation is not consistent.

This chapter is an introduction to contexts as terms with holes in the untyped and simply typed lambda calculus, to problems related to formalisation of contexts and to solutions for these problems. Also, this chapter establishes terminology related to contexts for the rest of the thesis. It is organised as follows. Section 2.1 is an introduction to contexts and context-related operations. In this section, different notions of context as encountered in the literature on rewriting, and their applications are described. Section 2.2 explains what a context formalisation comprises, and analyses the main problems in context formalisation. In this section, we define the notion of communication between a context and terms to be put into its holes. Communication is concerned with establishing intended variable capturing and passing on the effects of earlier α-conversion and β-reduction within the context to the terms filled into its holes. In Section 2.3 motivation for formalisation of contexts is given. In Section 2.4 an overview is given of existing context formalisms and their solution for the context-related problems is described. Section 2.5 gives a gentle introduction to our context calculus λc. Here, the main aspects of the context calculus λc are informally sketched. A formal description of the calculus

will be given in Chapter 3.

Later, in Chapter 6 typed lambda calculi with more expressive power than the simply typed lambda calculus, and contexts as terms and types with holes will be considered. The additional problems with and our approach to the formalisation of such generalised contexts will be discussed there.

2.1 Contexts in the lambda calculus

In lambda calculus contexts are commonly engaged in many arguments on the meta-level. In this section we treat the informal notion of context in the untyped and simply typed lambda calculus and list a few existing variations on the notions of context.

A context over λ-terms, or a (λ-)context for short, is basically a λ-term with some holes in it. Holes[1] are usually denoted by $[]$. For example, $(\lambda y.\,[])x$ is a λ-context.

A hole of a λ-context may be filled by a λ-term or by a λ-context. For example, filling the term xy into the context $(\lambda y.\,[])x$ results in the term $(\lambda y.\,xy)x$. Filling, for example the context $x(\lambda y.\,[])$ into the context $\lambda x.\,[]$ results in the context $\lambda x.\,x(\lambda y.\,[])$. In general, the resulting objects are denoted as $C[M]$ and $C[D]$ where C and D are contexts, M is a term and where $[\,]$ denotes the textual replacement of the hole(s) in C by M or D.

A peculiar side effect of filling holes of a context is variable capturing: when a term M or a context D is placed into the hole(s) of context C, some free variables of M or D may become bound by the binders of C. For example, the variable y, which is free in the term xy becomes bound by the binder λy of $(\lambda y.\,[])x$ in $((\lambda y.\,xy)x)$. This variable capturing is intended: by the choice for the names of the free variables in M and D, and the names for the binders of C one controls the variable capturing in the result of the filling. The variable capturing distinguishes filling of holes from substitution. Recall that substitution avoids such capturing by renaming problematic bound variables. For example, if we consider $[]$ as a variable for the sake of argument, the substitution $[\![[] := xy]\!]$ applied to $(\lambda y.\,[])x$ results in $(\lambda y'.\,xy)x$, where λy has been renamed into $\lambda y'$ in order not to bind the free y of xy.

Contrary to terms, contexts are not considered modulo α-conversion (in the name-carrying representation) nor are contexts subject to β-reduction. That means, for example, that $\lambda x.\,[] \neq_\alpha \lambda y.\,[]$ and $(\lambda x.\,x[])y \not\to_\beta y[]$. Furthermore, substitution is not defined on contexts. So, for example, $x[][\![x := y]\!] \neq y[]$. Because neither substitution, α-conversion nor β-reduction are defined on contexts, contexts are merely a notational convenience. That is, contexts are meta-elements of lambda calculus.

In the literature on rewriting, several variants of this simple view on contexts as terms with holes exist. The first possibility for variation is in the number of holes

[1] A hole is sometimes called an open place (cf. [Vri87]) or an open end (cf. [Bru78]), and it is sometimes denoted by $[\,]$ (cf. [Bar84]).

allowed in a context: precisely one, or many, including zero. The second possibility for variation is, in the case where many holes are allowed, in the way these holes are treated: as copies of the same hole, which therefore must be filled with the same term; as copies of different holes, which therefore may be filled with possibly different terms; or as combination of both treatments by distinguishing between holes and hole occurrences.

We list a few definitions of different notions of context as found in the literature, and explain when a particular notion of context is needed. We will ignore the differences in the particular version of lambda calculi (with or without function symbols or typing, for example) because these differences are irrelevant for the notion of context. In the definitions we adjust the notation and terminology to correspond with ours. All these contexts are considered on the meta-level of lambda calculus.

The definitions consist of two parts: the first part describes the structure of a context, and the second part defines filling of holes by terms. It is in the definition of filling that one sees how []'s are treated: as copies of the same hole or as copies of different holes.

i) In [Klo80, Vri87] contexts are terms with a unique hole occurrence. The next definition amounts to Definition 1.5 in [Klo80] and Definition 0.2 in [Vri87].

Definition 2.1.1 (Contexts 1)

(a) A context C is a term with one hole occurrence $[]$.

(b) Let C be a context and M a term. Then $C[M]$ is the result of filling that hole occurrence with M.

These contexts are used when one focuses on one position in the term $C[M]$ and the changes that occur at that position. Such contexts are used, for example, in the definition of the rewriting rules of lambda calculus. For example, the β-rewrite relation is defined[2] by

$$C[(\lambda x.\, M)\, N] \to C[M[\![x := N]\!]] \qquad (\beta)$$

where C is a context, and M and N are terms.

Another occasion where such contexts are used is in proofs by induction to the structure of a term.

ii) In [Bar84], contexts are terms with many hole occurrences (cf. Definition 2.1.18 in [Bar84]).

Definition 2.1.2 (Contexts 2)

[2] In fact, this is the only way to define β-rewriting by one rule schema. Without contexts one has to express that rewriting is a congruence, which then takes more words to describe it.

(a) A context is a term with some holes in it. More formally:

x is a context,
$[]$ is a context,
if C_1 and C_2 are contexts, then so are $C_1 C_2$ and $\lambda x.\, C_1$.

(b) If C is a context and M is a term, then $C[M]$ denotes the result of placing M in the holes of C.

These contexts are used when one considers many occurrences of the same subterm M in a term $N \equiv C[M]$. Such contexts are used, for example, in λI in the proof that a term M is solvable if and only if M has a normal form. The contexts of Definition 2.1.1 are a special case of these contexts.

iii) In [Oos94] contexts are terms with many holes which occur at most once. The following definition can be extracted from Definition 3.(4) of the same reference.

Definition 2.1.3 (Contexts 3)

(a) A context is a term containing holes $[]_1, \ldots, []_n$, where $n \geq 0$.

(b) If C is a context containing $[]_1, \ldots, []_n$, and $\vec{M}$ are n terms, the term $C[\vec{M}]$ denotes the result of replacing $[]_i$ in C by M_i, for $1 \leq i \leq n$.

These contexts are used when one focuses on simultaneous changes at a set of positions. For example, such contexts are used in the definition of development steps (cf. Definition 1.3.21 in [Oos94]):

$$C[l_1, \ldots, l_n] \multimap C[r_1, \ldots, r_n]$$

where $l_i \to r_i$ is a rewrite rule for $1 \leq i \leq n$. Implicitly, the existence of such a context C with $s \twoheadleftarrow_\beta C[l_1, \ldots, l_n]$ defines the notion of independent redex occurrences in s (cf. Definition 1.3.22, see also Examples 1.3.24 and 1.3.23).

iv) In [Ber78] contexts are terms with many holes which may occur many times. Holes are labelled and contexts are called multiple numbered contexts. The following definition is Definition 14.4.1 in [Bar84].

Definition 2.1.4 (Contexts 4)

(a) Multiple numbered contexts are defined as:

x is a multiple numbered context,
if $i \in \mathbb{N}$, then $[]_i$ is a multiple numbered context,
if C_1 and C_2 are multiple numbered contexts, then $C_1 C_2$ and $\lambda x.\, C_1$ are multiple numbered contexts.

(b) Let C be a multiple numbered context such that the holes of C are among $[]_{i_1}, \ldots, []_{i_n}$ with $i_1, \ldots, i_n \in \mathbb{N}$ and $i_1 < \ldots < i_n$. Let $\vec{M}$ be n terms. Then $C[\vec{M}]$ denotes the result of placing M_j in the hole $[]_{i_j}$ for $1 \leq j \leq n$.

These contexts are used when tracing many positions in the term $C[\vec{M}]$ along a rewrite sequence. For example, such contexts are used in the proof that computation of functions that are definable in lambda calculus is sequential rather than parallel (cf. [Ber78]). The contexts of Definitions 2.1.1 through 2.1.3 are all special cases of multiple numbered contexts.

In the simply typed lambda calculus, contexts are defined in similar ways. The differences are that []'s have each a type, that contexts are *well-typed* terms with holes with respect to the typing rules at hand, and that hole filling is required to be type-preserving.

Remark 2.1.5 With contexts being defined as *terms* with holes, the definition of filling holes by terms implicitly defines the operation of filling holes by contexts. Although filling of holes by contexts is analogous to filling of holes by terms, some precision is desirable, in particular when dealing with contexts according to Definition 2.1.4. For example, in Definition 2.1.4, one talks about 'a context over holes $[]_{i_1}, \ldots, []_{i_n}$' but there is no restriction that these holes actually have to occur in the context. For example, we may consider the context $D \equiv \lambda y.\, []_2$ as a context over $[]_2$ and $[]_3$. Filling this context into the context $C \equiv \lambda x.\, []_1$ results in the context $\lambda x.\, \lambda y.\, []_2$. Is this resulting context considered as a context over $[]_2$ and $[]_3$ or as a context over $[]_2$? Whatever one chooses for the answer to this question, for the sake of clarity an answer should be provided.

Note that the restriction that $[]_{i_1}, \ldots, []_{i_n}$ have to occur in 'a context over holes $[]_{i_1}, \ldots, []_{i_n}$' cannot be imposed. In that case, hole filling would be defined on a context with n holes (i.e. a context which actually contains n holes) and n terms. However, multiply numbered contexts are considered along a rewrite sequence $C[\vec{M}] \to_\beta C'[\vec{M}] \twoheadrightarrow \ldots$ and along a rewrite sequence a hole may disappear: for example, $((\lambda x.\, y)\, []_1)[z] \to y[z]$. Then, $y[z]$ would not be a valid expression, and thus, the set of expressions would not be closed under rewriting.

2.2 Context formalisation: problems and analysis

The main goal in this thesis is to formalise contexts. A formalisation of λ-contexts should ideally establish the following[3]:

- it should provide a means for representing contexts,

[3]The formalisation items that are listed above can be seen in any formalisation of a meta-level notion. For example, in explicit substitution calculi (see e.g. [ACCL91]) representation of substitutions, representation of substitution-related operations and their computation is provided, and β-reduction is defined on (representations of) substitutions.

- it should provide a means for representing and computing context-related operations, and
- it should extend the standard operations and relations to (the representations of) contexts: these transformations comprise substitution, rewrite relations and, in the case of a typed lambda calculus, also the typing relation.

These requirements for a formalism are not independent of each other. For example, the way α-conversion and β-reduction are defined on contexts will influence the syntax of expressions, including the representation of contexts. Still, there is plenty of room for variation in a context formalisation. For example, formalisms may differ in the way context-related operations are computed: by means of meta-operations or by means of rewrite rules; or formalisms may or may not allow contexts to be an argument or a result of a function. However, and on this point all context formalisms that we discuss in Section 2.4 agree, the most important part of the formalisation of contexts are the requirements listed above, and in particular, the extension of α-conversion and β-reduction to contexts. Hence, the requirements listed above can be understood as minimal requirements for a context formalisation.

We address two main problems in formalisation of contexts as terms with holes in the untyped and simply typed lambda calculus. The first problem concerns the combination of α-conversion and β-rewriting on contexts on the one hand, and hole filling on the other hand. The second problem concerns the preservation of the notion of context under β-rewriting. The problems that arise from typing, and in particular, from typing when holes are allowed in types too, will be addressed in Chapter 6.

Non-commutation of rewriting and hole filling

The major problem[4] in a naïve formalisation is that the standard rewrite relations do not commute with the new context reductions. Confluence is lost and, consequently, the corresponding equational theory is inconsistent. The non-commutation of β-reduction and hole filling with terms is demonstrated by the next example, where a representation for *hole filling*, *hf* (here seen as an operator), has been introduced, where hole filling is computed by *fill*-reduction and where β-reduction has naïvely been extended to contexts. This example will be used as the running example throughout the thesis. We will return to this example to show how the problem described here is solved in a particular formalism, including our context calculus. Furthermore, a similar example of non-commutation can be given for substitution or α-conversion instead of β-reduction.

Example 2.2.1 For the sake of this example we make the naïve formalisation of contexts precise. We define expressions and rewrite rules. Let expressions P, which

[4]This problem is generally recognised in the literature on context formalisation (see Section 2.4).

now represent both terms[5] and contexts, be defined as

$$P ::= x \mid [] \mid \lambda x.P \mid PP \mid hf(P,\, P).$$

Let β-reduction be generated by the rewrite rule

$$(\lambda x.P)Q \to P[\![x := Q]\!] \qquad (\beta)$$

and filling of holes by

$$hf(P,\, Q) \to P[Q]. \qquad (\mathit{fill})$$

Note that with expressions and hole filling defined as above, this formalism includes contexts of Definition 2.1.2. Let $C \equiv (\lambda y.[])x$ and $M \equiv xy$. Then

$$hf(C,\, M) \;\equiv\; hf((\lambda y.[])x,\, xy) \to_{\mathit{fill}} (\lambda y.xy)x \to_\beta xx$$

but

$$hf(C,\, M) \;\equiv\; hf((\lambda y.[])x,\, xy) \to_\beta hf([],\, xy) \to_{\mathit{fill}} xy \neq xx.$$

In the example, the reductions[6] end in different terms because in the first reduction the substitution $[\![y := x]\!]$, which emerged from the rewrite step in context C is applied to the term M, while in the second reduction the substitution is not applied to the term M, but only to the hole, which 'forgets' it. Note that the result of the first reduction is the intended one. Note also the subtle difference between denoting a hole filling $hf(P,\, Q)$ and its result $P[Q]$.

The problem is that, when α-conversion and β-reduction are defined on contexts, the interaction between a context and a term (or a context) to be put into its holes becomes more complex than plain variable capturing. We call this interaction between a context and a term (or a context) to be put into its holes *communication*. The following informal definition is meant to stipulate the way we use the word 'communication'.

Definition 2.2.2 (Communication) Let C be a context and M a term to be filled into a hole of C. Communication between (reducts of) C and M comprises establishing intended variable capturing and passing on imminent substitutions to M that emerge from earlier α-conversion and β-reduction within the context C.

Communication between a context C and a context D which is to be put into a hole of C is defined analogously.

Thus, in lambda calculus, where α-conversion and β-reduction are not defined on contexts, communication boils down to establishing intended variable capturing.

[5] It is interesting to see that by broadening our view to both terms and contexts, terms may be defined as contexts without holes.

[6] In the example we defined hole filling as a rewrite relation. Also, if we computed the hole filling by the meta-operation $P[Q]$, the two 'rewrite' sequences would still result in different terms.

Here, the intended variable capturing is easy to establish, as we have already mentioned at the beginning of Section 2.1, by the choice for the names of binders of C and for the names of free variables of M or D to be filled into the holes of C.

In lambda calculus we are not much aware of the communication because it is implicitly solved. However, in a context formalism communication will be visibly present. At the point of representing a (meta-)context as an expression of the formalism, the intended variable capturing will be fixed and it should remain preserved under rewriting.

Thus, in a context formalism, what is needed to solve the problem of communication is a way of denoting the intended bindings, which keeps track of α-conversion or β-reduction in the outer context and passes the effects of these reductions on to the terms (or contexts) replacing the holes. The effects of α-conversion or β-reduction in the outer context always result in a substitution: in the case of α-conversion, $\lambda x.\, P =_\alpha \lambda y.\, P[\![x := y]\!]$; and in the case of β-reduction $(\lambda x.\, P)Q \to_\beta P[\![x := Q]\!]$. The effects of many α- or β-steps may accumulate in one substitution $[\![\vec{x} := \vec{P}]\!]$. In this substitution, $\vec{P}$ may contain holes too. Such a substitution should pause at a hole until a term or a context is filled into this hole. The substitution should then be applied to the term or context filled into the hole.

For example, the second reduction of the example above can be repaired by explicitly keeping track of these α, β-changes and applying the resulting substitution to the term after hole filling:

$$hf((\lambda y.\,[])x,\, xy) \to_\beta hf([]^{[\![y:=x]\!]},\, xy) \to_{fill} (xy)[\![y := x]\!] = xx.$$

The problem of establishing communication in a context formalism is reduced to the encoding of this substitution.

Preservation of the notion of context within a formalism

In addition to solving the communication problem, there is also the issue of the notion of context *within* a formalism. At this point we distinguish between the notion of context on the meta-level and the notion of (representation of) context within a formalism.

In a context formalisation, the structure of (the representation of) a context regarding the number and treatment of holes has to be preserved under rewriting. For example, contexts with exactly one occurrence of $[]$ are not preserved under rewriting, since this hole occurrence may be duplicated under β-reduction, e.g. $(\lambda x.\, xx)[] \to_\beta []\,[]$. This counterexample shows that contexts of Definitions 2.1.1 and 2.1.3 are not preserved under rewriting: in the reduct $[]\,[]$ there are two occurrences of $[]$ (which conflicts with Definition 2.1.1(i)) which should be considered as occurrences of the same hole and accordingly, should be filled by the same term (which conflicts with Definition 2.1.3(ii)). One easily sees that contexts of Definition 2.1.2 and 2.1.4 are preserved under rewriting.

Note however that this does not mean that contexts of Definition 2.1.1 and 2.1.3 cannot be formalised. These contexts can be formalised, but within a formalisation they will be represented by contexts of Definition 2.1.2 and 2.1.4, respectively.

In our opinion the most natural notion of context *on the meta-level*, where α-conversion and β-reduction are not defined on contexts, is as defined by Definition 2.1.3. Each [] in such a context is characterised by its position, by the binders in whose scope it lies, and by its type (if typed). Because two []'s positioned at different positions in general lie in the scope of different variables, it is the most natural way to consider these []'s as different holes. However, when (meta-)contexts are represented in some context formalism, where α-conversion and β-reduction are defined on contexts, multiple numbered contexts (Definition 2.1.4) should be used, because holes can be duplicated under rewriting.

Intermezzo 2.2.3 In this section we have shown that a naïve formalisation of contexts leads to an inconsistent system. There is however a special case where the rewrite relations can naïvely be extended to contexts to form a consistent system. This case pertains to contexts, whose structure is preserved under rewriting (Definition 2.1.2 or 2.1.4) and which may be filled only by closed terms or by closed contexts. If a term M to be put into a hole of C is closed, then there is no communication between M and C. Hence, filling of holes boils down to 'substitution' of []'s. Using the Substitution lemma 1.2.17, it holds that $C \to_\beta C'$ if and only if $C\,[\![[] := M]\!] \to_\beta C'[\![[] := M]\!]$:

$$
\begin{array}{lll}
C \to_\beta C' & & \\
\quad \Leftrightarrow & \exists D, P, Q \text{ with } C \equiv D[(\lambda x.\, P)\, Q] \to_\beta D[P[\![x := Q]\!]] \equiv C' & \\
\quad \Leftrightarrow & \exists D, P, Q \text{ with} & \\
& \quad C[\![[] := M]\!] & \\
& \qquad \equiv & D[(\lambda x.\, P)\, Q][\![[] := M]\!] \\
& \qquad = & D\,[\![[] := M]\!]\,[(\lambda x.\, P[\![[] := M]\!])\, Q[\![[] := M]\!]] \\
& & \text{because } x \notin \mathrm{FVAR}(M) \\
& \qquad \to_\beta & D\,[\![[] := M]\!]\,[P\,[\![[] := M]\!]\,[\![x := Q[\![[] := M]\!]]\!]] \\
& \qquad = & D\,[\![[] := M]\!]\,[P[\![x := Q]\!]][\![[] := M]\!] \\
& & \text{by the Substitution lemma and } x \notin \mathrm{FVAR}(M) \\
& \qquad = & D[P[\![x := Q]\!]][\![[] := M]\!] \\
& \qquad \equiv & C'\,[\![[] := M]\!]\,.
\end{array}
$$

Alternatively, instead of computing the hole filling by substitution (which is a meta-operation), one can easily define a context calculus, where the hole filling restricted to the filling with closed terms and closed contexts is computed by a rewrite relation:

- use hole variables $h_1, \ldots, h_n$ instead of []: if C is a context over n holes, then represent the context as $C[h_1, \ldots, h_n]$,
- represent contexts as abstractions over hole variables $\lambda\vec{h}.\, C[\vec{h}]$ and hole filling as applications $(\lambda\vec{h}.\, C[\vec{h}])\,\vec{M}$, and
- compute hole filling by β-reduction: $(\lambda\vec{h}.\, C[\vec{h}])\,\vec{M} \to C[\vec{h}][\![\vec{h} := \vec{M}]\!] \equiv C[\vec{M}]$.

Rewriting in such a system is confluent, because of the confluence property of β-reduction.

The property that β-rewriting commutes with hole filling on such contexts is used in the definition of rewrite steps in higher-order rewriting in [Oos94]. Recall that on terms s and t of a PRS $\mathcal{H}$, we have $s \to_{\mathrm{i}} t$ if and only if there are a context C and a rewrite rule (i) : $l \to r$ such that $s \twoheadleftarrow_\beta C[l]$ and $t \twoheadleftarrow_\beta C[r]$. Because l and r are closed terms, one can be more precise about the choice for the context C by taking the β-normal form as the representative. Here, 'taking the β-normal form of a context' means that β-reduction is defined on contexts. The property that β-rewriting commutes with hole filling on contexts and closed terms is used in the proof of the Prism theorem in [Oos95], which in turn is used to prove that orthogonal PRSs are confluent.

Intermezzo 2.2.4 We briefly consider (first-order) term rewrite systems (TRSs, see for example [Klo92]) and formalisation of contexts in TRSs. Because there are no binders in TRSs, communication between a context and terms to be put into its holes is void. Then, rewriting can directly be extended to contexts, provided that rewriting preserves the notion of context (the number of holes and their occurrences) in question. Moreover, the notion of context is preserved under rewriting if the rewrite rules are linear and non-erasing. A rewrite rule is linear if each variable occurs at most once in the left-hand side and at most once in the right-hand side of the rule. A rewrite rule is non-erasing if all variables that occur in the left-hand side occur also in the right-hand side of the rule.

2.3 Motivation for formalisation of contexts

The starting point of our research has been De Bruijn's calculus of segments, which was proposed in the context of the family of proof checkers Automath. From a broader perspective, the increasing interest in contexts has its motivation from many directions, as diverse as modelling programs and program environments, operational semantics and dealing with anaphora in natural language representation. In all these cases there is a need for manipulating contexts on the same level as expressions.

The applications of contexts we discuss here go beyond the applications of different notions of context that we have discussed in Section 2.1. We give examples of three different applications of contexts.

Examples from programming

We discuss two examples of the usage of contexts in the realm of programming.

In the field of programming, program transforming techniques have been developed with the aim of optimisation. The main idea is to transform programs written by programmers into less structured but more efficient programs, while preserving their meaning. An example of a program transformation is the transformation of a recursion statement into an equivalent iteration statement (cf. [HL78]).

In general, program transformations are expressed by means of rewrite schemes. Such schemes are easily defined by using contexts. However, a naïve approach to stating such schemes may lead to incorrect program transformations. The following example is given in [PE88]: distribution of a context over an if-statement, expressed by the scheme

$$C[\texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ N] \Leftrightarrow \texttt{if}\ B\ \texttt{then}\ C[M]\ \texttt{else}\ C[N],$$

can be applied incorrectly:

$$\begin{array}{l}\texttt{let}\ p = \texttt{false}\ \texttt{in}\ \texttt{if}\ p\ \texttt{then}\ 1\ \texttt{else}\ 2\\ \not\Leftrightarrow \texttt{if}\ p\ \texttt{then}\ \texttt{let}\ p = \texttt{false}\ \texttt{in}\ 1\\ \qquad\quad \texttt{else}\ \texttt{let}\ p = \texttt{false}\ \texttt{in}\ 2.\end{array}$$

Intuitively, the first statement 'reduces' to 2, while the second statement 'reduces' to $\texttt{if}\ p\ \texttt{then}\ 1\ \texttt{else}\ 2$.

When formulating rewrite schemes with context propagation one has to take into account the bindings between the binders of C and the free variables of B, M and N.

Thus, in program transformation one considers statements as expressions and the surrounding code as contexts. Alternatively, one could consider programs as expressions and environments as contexts. A relevant operation involving programs and environments is evaluation of programs in an environment, where a similar binding occurs between for example system calls that are defined in the environment and used in a program. In this case, environments are seen as pairs of (variable/identifier, value), that is, as substitutions $[\![\vec{x} := \vec{M}]\!]$. Such a substitution can be represented by the context $C \equiv (\lambda\vec{x}.\,[])\,\vec{M}$. When representing an evaluation of a program N in an environment C, the hole filling is immediately followed by the reduction of the redexes in the context: $((\lambda\vec{x}.\,[])\,\vec{M})[N] \to N[\![\vec{x} := \vec{M}]\!]$. Formalisation of environments as contexts has been studied in [SSB99, SSK01, LF96].

Remark 2.3.1 One may also consider programs and procedures as expressions, and modules as contexts. The operations involving programs, procedures and modules are execution of a program using modules, or selection of a procedure from a module. In this case, one cannot achieve the same expressive power in lambda calculus as in a programming language: sometimes procedures are mutually dependent, and this kind of dependence cannot be expressed by lambda terms or contexts. A formalism dealing with such procedures and modules has been studied by e.g. J.B. Wells and R. Vestergaard in [WV00].

An example from proof checking

We give an example from type theory, where λ-contexts are used for the representation of mathematical concepts, in the realm of proof checking. This example illustrates the use of variables for contexts, which then also can be abstracted.

A context that is suitable for reasoning about reflexive relations, and hence in a way, that is suitable for representing the notion of reflexive relation, could be the following:

$$\lambda A : Set.\, \lambda R : (A \to A \to Prop).\, \lambda rfl : (\forall x : A.Rxx).\, [].$$

Let us call this context $\underline{refl}$. Then, an argument on reflexive relations, say a piece of mathematical text $text$, can be performed within this context, via hole filling:

$$\underline{refl}[text].$$

Here, $\underline{refl}[text]$ denotes the result of the meta-operation of hole filling. In $text$ the identifiers A, R, rfl can then be used. In a larger piece of text this can happen more than once. Say in a proof term

$$P(\underline{refl}[text_1], \underline{refl}[text_2], \underline{refl}[text_3]).$$

An efficient representation, without the need to repeat $\underline{refl}$, could then be given as:

$$(\lambda c.P(hf(c, text_1), hf(c, text_2), hf(c, text_3)))\underline{refl}$$

with hf as a hole-filling operator (as opposed to the hole-filling meta-operation whose result is denoted using $[\,]$) as in Example 2.2.1. Hole filling $hf(c, text_i)$ will be computed eventually by a hole-filling rewrite rule after $\underline{refl}$ is replaced for the context variable c. By providing such a hole-filling operator and a hole-filling rewrite rule we have a means for denoting hole filling and postponing its computation.

This example shows that contexts can effectively be employed for representing mathematical structures and furthermore, that for an efficient representation, also functions ranging over contexts as well as a means for delayed computation of hole filling is necessary. Functions over contexts implement an abbreviation mechanism for contexts (see also Remark 1.2.16 about abbreviation mechanism for terms in lambda calculus) and a delayed hole filling supports it.

The use of the context $\underline{refl}$ that we indicated is also a typical example of a *segment*, according to N.G. de Bruijn [Bru78]. A technical treatment of segments using our calculus λc will be given in Chapter 5.

An example from linguistics

The aim of natural language semantics is to give a method by which (a part of) natural language can be translated into a logical formalism. Often, a type theory is used as the logical formalism. The formalisation method is based on the compositionality principle: the representation of the whole text is a function of the representations of the pieces of the text.

We illustrate how contexts can be used to represent pieces of text. We take the following text and translate it, sentence by sentence, as contexts.

A man walks. He talks.

Here, the pronoun *he* in the second sentence is 'bound' to the declaration *a man* in the first sentence. When sentences are translated into a formalism one by one, this binding has to be preserved. This kind of binding is called dynamic binding in the natural language semantics.

In lambda calculus, the sentences can be represented as

$$S_1 \equiv \lambda x^{man}.W(x) \wedge [] \qquad \text{and} \qquad S_2 \equiv T(x) \wedge [],$$

where $\wedge$ denotes a language connective. The representation of the whole text is obtained by placing the second context S_2 in the hole of the first context S_1:

$$S_1[S_2] \;\equiv\; (\lambda x^{man}.W(x) \wedge [])[T(x) \wedge []] \;\equiv\; \lambda x^{man}.W(x) \wedge T(x) \wedge [].$$

Formalisation of such contexts is the subject of for example [KKM99].

2.4 Contexts in context

The study and formalisation of contexts has been the subject of various papers; in this section we list a few from the problem areas mentioned in Section 2.3. At the end of this section some related work is mentioned.

The context formalisms considered in this section agree on minimal requirements for formalisation of contexts, which were stated at the beginning of Section 2.2. Furthermore, these formalisms can be analysed and compared to each other by considering the formalisation dimensions given below. These dimensions can be understood as design choices, which depend heavily on the applications of a particular context formalism. Some dimensions are not appropriate for all formalisms.

- Is a formalism a method for context representation or a context calculus?

  A formalism is a context *calculus* if the context-related operations are represented as rewrite relations, as opposed to as a meta-operation. This is analogous to lambda calculus, which formalises the notion of function (it might as well be called a 'function calculus'). In lambda calculus a function $\lambda x.M$ can be applied to an argument N, viz. $(\lambda x.M)N$, and its computation (represented by β-reduction) can be postponed.

  Accordingly, a formalism which represents context-related operations as meta-operation is called a *method* for context representation.

- How does a formalism internalise communication?

  There seem to exist two main streams in tackling communication: the formalisms that employ an explicit substitution calculus, and the formalisms that employ higher-order rewriting techniques. In the first kind, holes are labelled by substitutions. In the second kind, communication uses the same technique that is used in the definition of rewrite step in higher-order rewriting: we will return to this comparison later in Section 3.5.1.

- What kind of transformations are defined on contexts?

  The transformations considered are the context-related operations, and the standard lambda calculus transformations, namely α-conversion and β-reduction. In some formalisms, β-reduction is conditional.

- Which properties does a formalism have?

  The most natural properties considered are confluence, strong or weak normalisation, and the subject reduction property in typed systems. In the overview of the literature on contexts given below, we will concentrate on the properties that are verified for a formalism.

- Is the formalism typed or not? Does the typing include dependent types?

- Which notion of context can be represented in a formalism? Is the notion of context fixed or not?

- Is the formalism defined over a specific signature or over an arbitrary signature?

By following these dimensions, some of the context formalisms discussed in the remainder of this section are compared in Table 2.1. We will return to these context formalisms and the comparison of the table for a more technical comparison in Chapter 4 or Chapter 5. Note that not all the context formalisms discussed below are included in the table: this is because the omitted context formalisms consider different context-related operations and use different methods in formalisation than those mentioned above.

We now give a brief description of a number of existing context formalisms.

In [Bru78], N.G. de Bruijn introduced a lambda calculus extended with incomplete terms of a special form, called *segments*. The purpose of segments was facilitating definitions and manipulation of abbreviations in Automath (see [NGdV94]). Technically, segments can be characterised as contexts with precisely one hole at the end of the spine[7]. The segment calculus included means for representing segments, variables over segments and abstraction over segments. In [Bal86, Bal87] H. Balsters gave a simply typed version of the segment calculus and proved confluence and subject reduction. In [Bru91], N.G. de Bruijn described a typed lambda calculus with telescopic mappings. Telescopes are segments consisting only of abstractions, which can be used for representing mathematical structures. Telescopic mappings are functions ranging over telescopes. The aim of the paper was not to give a calculus (instead, N.G. de Bruijn refers to the segment calculus), but to set out the main ideas on telescopes and their usage on the level of meta-language. Although the paper is positioned in a typed lambda calculus, the ideas are applicable to any rewrite system with binders.

With the goal of optimisation of interactive proof checking, L. Magnusson (see [Mag96]) presented an algorithm for incomplete proofs. The algorithm is designed for Martin–Löf's type theory with explicit substitutions and it is used in

[7]The end of the spine of a λ-term is the leftmost leaf of that λ-term in tree notation.

citation	method/ calculus	notion of contexts	comm.	transformations				properties		signature
				α	β	hf	*comp*	CR	SN	
[Bru78]	calculus	Def.2.1.1$^-$	by expl.subs.-like r.r.					yes	?	$\lambda, \lambda^{\rightarrow}$
[HO98]	calculus	Def.2.1.1	expl.subs.$^-$	yes	r.r.$^-$	r.r.	no	yes	?	$\lambda^{\rightarrow}$
[San98]	method	Def.2.1.2	HO-style	r.r.	r.r.	meta-ops		yes	yes	$\lambda^{\rightarrow}$, any
[Mas99]	method	Def.2.1.2	expl.subs.	r.r	r.r.	meta-ops		yes	?	λ
[SSB99]	calculus	Def.2.1.1	expl.subs.	r.r.	r.r.	r.r.$^-$		yes	yes	$\lambda^{\rightarrow}$
[SSK01]	calculus	Def.2.1.4	expl.subs.	r.r.	r.r.	r.r.	r.r.	yes	yes	$\lambda^{\rightarrow}$
[KKM99]	calculus	Def.2.1.4	HO-style	r.r.$^-$	r.r.$^-$	r.r.	r.r.	yes	yes	$\lambda^{\rightarrow}$
[Bru91]	method	Def.2.1.1$^-$	HO-style	not considered						arb.dep.typ.
λc	calculus	Def.2.1.4	HO-style	r.r.	r.r.	r.r.	r.r.	yes	$\lambda c^{\rightarrow}, \lambda c^{s}$	λ, any

Legend:

arb.	=	arbitrary
dep.typ.	=	dependent types
expl.subs.	=	explicit substitutions
r.r.	=	rewrite relation
P$^-$	=	a restricted notion of P
?	=	unknown

Table 2.1: Contexts in context

the proof editor ALF. The unfinished parts of a proof are denoted by placeholders, which are assigned a type and a local context. When filling in a new part of a proof into a placeholder, it is sufficient to check the new part. With the goal of representing incomplete proofs and supporting incremental proof development and higher-order unification, C. Muñoz presented in [Muñ97] a name-free explicit substitutions calculus with dependent types and with meta-variables ranging over the missing parts of an expression. In both formalisations, filling of the missing parts of an expression is an operation, which 'commutes' with (a refined version of) β-reduction.

With the development of programming languages in mind, M. Hashimoto and A. Ohori (see [HO98]) proposed a typed context calculus, which is an extension of the simply typed lambda calculus. The type system specifies the variable-capturing nature of contexts with one hole using α-sensitive interface variables. The relations of β-reduction and hole-filling reduction are combined, under the restriction that no β-steps are allowed within a context. With the aim of building a theory of separate compilation and incremental programming, S.-D. Lee and D. Friedman (see [LF96]) designed a schema for enriching lambda calculus with contexts. They employ contexts for modelling program modules and their calculus for modelling module linking. In their calculus the binders in contexts are treated as identifiers whose binding scope is by compilation extended to objects filled into the holes. Computation is performed by β-reduction and additional compilation rules. M. Sato, T. Sakurai and R. Burstall (see [SSB99]) defined a simply typed lambda calculus with first-class environments. The calculus is provided with operations for evaluating expressions within an environment and includes environments as function arguments. It is basically an explicit substitution calculus with substitution variables and functions over such variables. This calculus was later used to formalise contexts as first-class citizens in [SSK01].

For dealing with contexts in operational semantics, D. Sands (see [San98]), contributing the idea to A. Pitts, proposed a representation of contexts with function variables for holes, meta-abstractions over variables to be captured for terms to be filled into the holes, and substitution of hole variables for hole filling. Using this representation, the operation of hole filling is freely combined with β-reduction. With the same motivation, I.A. Mason (see [Mas99]) extended the syntax of lambda calculus with notations for holes labelled by a substitution. He introduced two notions of variable replacement, weak and strong substitution, which differ in the behaviour at hole labels. Weak substitution is used for α- and β-reduction, and for filling holes with terms. Strong substitution is used for filling holes with contexts. Hole filling is defined as an operation, which 'commutes' with β-reduction.

With the purpose of modelling binding mechanisms in natural language, M. Kohlhase, S. Kuschert and M. Müller (see [KKM99]) introduced dynamic lambda calculus as an extension of the simply typed lambda calculus with declarations. In their approach the scope of binders sometimes extends the textual scope of a sentence. Declarations are α-sensitive and β-reduction is not defined on declarations. In addition to types, expressions are provided with modality, which describes their variable binding power.

Other related work. Other contributions to the formalisation of contexts have been made by C.L. Talcott [Tal91] and S. Kahrs [Kah93].

Related work concerns the fields of research on explicit substitution calculi (see for example [ACCL91]), higher-order rewriting (see for example [Klo80, MN98, Oos94, Raa96, Wol93]), higher-order syntax (see for example [DPS97]), higher-order unification (see for example [Mil91]), and linguistics (see for example [GS91, Kam81, Ran91, Zee89]). These fields either provide methods for solving communication problems (like for example explicit substitution calculi or higher-order rewriting), or deal with communication problems in an alternative way (like for example linguistics).

2.5 Our approach

Although emerging from different fields of research, with different motivations, the problem of formalising contexts and communication can be tackled uniformly. Our context calculus λc can serve as a uniform framework for representing different kinds of contexts. It is an extension of the lambda calculus with facilities for representing contexts and context-related operations such as filling the holes of a context by terms or by contexts, and establishing the (explicit) communication.

The context calculus λc is a *calculus*. That means that contexts can freely be manipulated on the object level: contexts may be an argument or a result of a function; and α-conversion and β-reduction as well as filling of holes is computed *within* the calculus, as opposed to computing these transformations by meta-operations. Such a calculus, with functions ranging over contexts and a means for denoting hole filling and delaying its computation, is necessary for applications in proof checking (segment calculus) and in linguistics, as can be seen in the previously mentioned work of N.G. de Bruijn, H. Balsters and M. Kohlhase et al.

Such a treatment of contexts is accomplished by giving contexts a functional representation. That is, a context is seen as a function of (the contents of) its holes, and accordingly, it is represented by λ-abstracting the hole variables. A functional representation of contexts is also present in the work of M. Hashimoto and A. Ohori.

In the context calculus λc, computing communication is handled separately from filling holes. We do so because, as can be seen from the problem analysis in Section 2.2, formalising communication poses a challenge of its own. With communication computed separately, filling of holes by terms or contexts is reduced to replacement of hole variables without any communication.

In the context calculus λc, communication is implemented using a technique which allows us to control the passing of variable bindings. This regards not only the intended variable capturing but also passing on the effects of earlier α-conversion and β-reduction within the context. There is an analogy to techniques developed in higher-order rewriting (see for example Aczel, J.W. Klop [Klo80], T. Nipkow [Nip93], V. van Oostrom & F. van Raamsdonk [OR93]), and in the field of higher-order abstract syntax (see for example F. Pfenning & C. Elliott [PE88], J. Despeyroux, F. Pfenning & C. Schürmann [DPS97]), where variable capturing

is accomplished by a substitution calculus. We will explain this analogy in Section 3.5.1. Similar techniques are applied in the work of D. Sands which we have already mentioned. Our technique gives holes as well as the expressions that are to be filled in, a functional representation.

Indirectly, by an explicit representation of communication, we distinguish between a term to be put into a hole of a context and a term itself. From the conceptual point of view, this distinction is natural: if a term is to be filled into a hole, the free variables of this term that will be captured after the filling are already considered bound. For example, no substitution for these variables is allowed. This is in contrast to the free variables of an arbitrary term. Analogously, we distinguish between a context and a context to be filled into a hole of an another context. We will call the terms and the contexts to be filled in *communicating terms* and *communicating contexts*, respectively. Note that communication roles are fixed in advance.

In our approach we also distinguish between filling holes by terms (to be called *hole filling*) and filling holes by contexts (to be called *composition*). Technically, the difference between these operations is, in addition to the difference in the objects that are placed into the holes (communicating terms vs. communicating contexts), in the resulting object: a λ-term, in the case of hole filling; and a λ-context, in the case of composition. In our view of a context as a function over its holes, the distinction between hole filling and composition is comparable, up to communication, to the distinction between a function application and a composition of functions. The result of applying function f to a value a results in a value $f(a)$; this value cannot be further applied to another value. The result of 'applying' function f to function g is the composition $f \circ g$; this composition can in turn be applied to a value. Although the distinction between hole filling and composition is not prominently present in lambda calculus, it is rather natural, as witnessed by the example of context application in linguistics.

The power of our calculus is its expressivity, which is achieved by on the one hand a flexible syntax, and on the other hand the possibility of term-formation restrictions within the framework. The syntax allows for a first-class treatment of contexts by having explicit abstraction over context variables and free context manipulation. Term-formation restrictions are implemented by typings. Via the choice of an adequate typing different notions of context can be represented within λc. Although λc is defined as an extension of lambda calculus, our approach to formalisation of contexts can be applied to an arbitrary rewrite system with or without binders. Last but not least, the calculus can directly be translated to lambda calculus. Bearing this in mind, we perceive λc as a comfortable level of abstraction for dealing with contexts as first-class objects.

The context calculus has been included in Table 2.1, where it can informally be compared to other existing context formalisms. In the remainder of this section we explain informally the main aspects of the context calculus. In Chapter 6 we will extend the explanation of our approach to formalisation of contexts as terms and types with holes in the typed lambda calculi of Barendregt's lambda cube.

Contexts. A context will be considered as a function ranging over the possible

contents of its holes. For this reason, in the context calculus hole variables $h, g, k, \ldots$ are introduced and contexts are represented as functions over (one or many) hole variables. The abstractor[8] for hole variables is denoted by δ_n, where $n \in \mathbb{N}$ is the number of variables which δ_n binds.

Communication. At first sight, it seems natural to use explicit substitutions (see for example [ACCL91], or [Blo99] where Pure type systems with explicit substitutions have been studied) for communication, by for example labelling holes with a substitution, viz. $[]^{\sigma}$. This idea can be found, as already mentioned, in for example the work on contexts of L. Magnusson [Mag96], C. Muñoz [Muñ97], I.A. Mason [Mas99] and M. Sato et al. [SSB99]). In the present paper it is our objective to reduce the whole matter of context manipulation to the very basic and well-understood notions of λ-abstraction and β-reduction. An explicit substitution calculus could then be used to eliminate β-reduction again, for example with the purpose of giving an efficient implementation. At this point we think it is profitable to separate the two issues.

So we take a basic, lambda-calculus-like approach, and solve the problem of encoding communication by using the fact that in lambda calculus substitution emerges as the result of a β-step: $M\,[\![x := N]\!] \leftarrow_{\beta} (\lambda x.\, M)\,N$. Since it is convenient to use multiple substitutions as communication may involve substitution of many variables, we will introduce new constructors $\Lambda_n \_ . \_$ for multiple abstraction of n variables and $\_\langle \_, \ldots, \_ \rangle_n$ for multiple ($n+1$-ary) application, together with a multiple version ($\textcircled{m}\beta$) of the β-rule. This is illustrated by the following example.

Example 2.5.1 The second, problematic reduction in Example 2.2.1 now becomes, in a reverse order of the steps (the hole-filling constructor hf is still auxiliary, indices are implicit and h is a hole variable):

$$\begin{array}{lll} x\,x & = & (x\,y)[\![y := x]\!] \\ & \leftarrow_{\textcircled{m}\beta} & (\Lambda y.\, x\,y)\,\langle x \rangle \\ & \leftarrow_{\mathit{fill}} & hf\,(h\,\langle x \rangle,\ \Lambda y.\, x\,y) \\ & \leftarrow_{\beta} & hf\,((\lambda y.\, h\,\langle y \rangle)\,x,\ \Lambda y.\, x\,y), \end{array}$$

where the last term shows the new representations of the hole[9] $(h\,\langle y \rangle)$ and of the *communicating term* $(\Lambda y.\, x\,y)$.

In general, communicating terms and, in the case of composition, communicating contexts are represented as multiple abstractions over variables that will become bound by the binders of the context where they will eventually be placed. Hence, the functional representation of communicating terms and communicating contexts: such terms and contexts are functions over communication. When a communicating term is placed into the hole, communication can be computed by applying a generalised form of the β-rule (which involves a simultaneous substitution)

$$(\Lambda x_1, \ldots, x_n.\, U)\,\langle V_1, \ldots, V_n \rangle \;\rightarrow\; U[\![x_1 := V_1, \ldots, x_n := V_n]\!], \qquad (\textcircled{m}\beta)$$

[8]The symbol δ is used also by M. Hashimoto and A. Ohori for abstracting hole variables, but only as a 'unary' abstractor.

[9]The notation $\langle\,\rangle$ in the representation of holes is also used by D. Sands.

recovering the binding intention and passing the changes. In such a representation, the representation of communication is localised to the representation of holes and communicating terms (or contexts), and as such, it is easier to handle.

Note that the constructor $\_\langle \_, \ldots, \_ \rangle_n$ is a $n+1$-ary function symbol, where the first argument is placed in front of the brackets. Such notation emphasises the special meaning of the first argument with respect to the other arguments: the first argument is a hole variable or a communicating term, while the arguments between the brackets represent a delayed communication.

Hole filling and composition. With communication localised around holes and communicating terms and communicating contexts, hole filling and composition reduce to (capture-avoiding) replacement of hole variables and formation of the result.

Since we represent contexts as functions over holes (i.e. as abstractions over hole variables), hole filling simply boils down to (multiple) β-reduction. Thus, in our representation of Example 2.2.1 we get

$$(\delta h.\, \lambda y.\, h\langle y\rangle\, x)\lceil \Lambda y.\, x\, y\rceil \to_{\mathit{fill}} (\lambda y.\, (\Lambda y.\, x\, y)\langle y\rangle)\, x,$$

where $\_\lceil \_ \rceil$ denotes a hole-filling constructor. In general, a context representation may be a function over many holes and consequently, hole filling may involve filling many holes simultaneously, viz.

$$(\delta_n h_1, \ldots, h_n.\, U)\lceil V_1, \ldots, V_n\rceil_n \to U\,[\![h_1 := V_1, \ldots, h_n := V_n]\!]\,. \qquad (\mathit{fill})$$

Here, the number of holes (i.e. the index of δ_n) equals the number of arguments between the brackets (i.e. the index of $\_\lceil \_, \ldots, \_\rceil_n$). Note that the arity of $\_\lceil \_, \ldots, \_\rceil_n$ is $n+1$.

Also composition may involve filling many holes simultaneously. This explains the need for composition operators $\circ_n$ for arbitrary n. However, the rewrite relation of composition is more complicated than in the case of hole filling: in the formation of the resulting contexts composition includes some shifting of abstractions. This is explained by the following example of a binary composition.

Example 2.5.2 Let $C \equiv \lambda x.\, []$ and $D \equiv x(\lambda y.\, [])$ be two λ-contexts. Then the composition of the two results in the λ-context $\lambda x.\, x(\lambda y.\, [])$. Note that the hole of the result of the composition is the 'lifted' hole of D, which potentially binds the variable x as well as the variable y.

In the context calculus, these λ-contexts are represented as

$$C_c \equiv \delta g.\, \lambda x.\, g\langle x\rangle \quad \text{and} \quad D_c \equiv \delta h.\, x(\lambda y.\, h\langle y\rangle).$$

Because the second context is going to be put into the hole of C_c, it is provided with means of communication: the preamble Λx and 'lifted' hole $h\langle x, y\rangle$ adapted for this purpose, viz.

$$D'_c \equiv \Lambda x.\, \delta h.\, x(\lambda y.\, h\langle x, y\rangle).$$

The composition puts the second context into the hole, and moves the abstraction δh to the beginning of C_c, so that the whole becomes an abstraction over the 'lifted' hole h of D_c. The composition rewrite step should result in

$$C_c \circ D'_c \to_\circ \delta h.\, \lambda x.\, (\Lambda x.\, x\,(\lambda y.\, h\,\langle x, y\rangle))\,\langle x\rangle,$$

where $\circ$ is the composition constructor in λc. Note that by performing the ensuing communication step this term reduces to $\delta h.\, \lambda x.\, x\,(\lambda y.\, h\,\langle x, y\rangle)$, which is a representation of the resulting composition in lambda calculus.

The $\circ$-step of the example is an instance of the binary-composition rewrite rule:

$$(\delta g.\, U) \circ (\Lambda u_1, \ldots, u_n.\, \delta h.\, V) \to\ \delta h.\, U[\![g := \Lambda u_1, \ldots, u_n.\, V]\!] \qquad (\circ)$$

where δh is shifted to the beginning of the reduct (after the variable h has been renamed if it occurs free in U). In the example a binary composition was used. In general, if a context representation is a function over n holes, the composition $\circ_n$ will involve $n+1$ contexts: one outer context and n contexts that are filled into the holes of the outer context. The resulting context is a context over the holes of the n contexts; hence, the composition will shift the hole abstractions of all n contexts to the beginning of the reduct.

Note that the constructors $\_\lceil\_, \ldots, \_\rceil_n$ and $\_\circ_n\_, \ldots, \_$ are $(n+1)$-ary function symbols, where the first argument is placed in front of the brackets in order to emphasise its special role: the first argument is the context whose holes are to be filled with the arguments between the brackets.

Framework. In the context calculus the building blocks can freely be combined to form λc-terms: variables, abstractions, applications and compositions. If a context contains many occurrences of [], they may be given the same name, like for example the occurrences of the hole h in the λc-term $\delta h.\, \lambda x.\, (h\,\langle x\rangle)\,(h\,\langle x\rangle)$. If a context contains many holes, they can be represented by different hole variables, like for example the holes h and g in $\delta h, g.\, \lambda x.\, (h\,\langle x\rangle)\,(\lambda y.\, g\,\langle x, y\rangle)$. An alternative representation is $\delta h.\, \delta g.\, \lambda x.\, (h\,\langle x\rangle)\,(\lambda y.\, g\,\langle x, y\rangle)$, where the holes should be filled sequentially. Last but not least, the calculus may include variables over contexts and functions ranging over contexts, witnessing the true first-class treatment of contexts.

In λc different notions of context can be represented. However, considering a calculus with contexts of a specific form, a criterion for well-definedness of such a calculus is of course that that specific form of contexts is preserved under transformations such as substitution, α- and β-rewriting, hole filling and composition.

Typing. The flexibility of the framework can be controlled by *typing*, that is, by restricting the λc-term formation. The aim of these restrictions is to gain more control over the form of λc-terms. Typing works in λc like typing does in lambda calculus. In a typed lambda calculus, each variable has a type and term formation is led by a set of typing rules. Analogously, a set of typing rules controls the λc-term formation. By means of typing, λc-terms can be restricted to representations of simply typed terms and contexts, λc-terms can be restricted to representations

of contexts with only one hole, variables in abstractions can be ensured to match their arguments, or the whole context calculus λc can be restricted to a subset of term constructors and rewrite rules, for example. Examples of typings for λc will be given in Chapters 4 and 5.

Parametrisation over signature. In this thesis, we focus on our approach to formalisation of contexts in lambda calculus. However, this approach combines well with an arbitrary rewrite system. This will be shown in Section 3.4. An arbitrary rewrite system can be extended with hole variables, communication abstractor and applicator Λ and $\langle\rangle$, hole-variables abstraction δ, hole-filling symbol $\lceil\rceil$, and composition $\circ$, and the rewrite rules $(\textcircled{\scriptsize m}\beta)$, (fill) and $(\circ)$. These rewrite rules will not interfere with the rewrite rules of the rewrite system because of the disjoint signature. Of course, the properties of such an extension of an arbitrary rewrite system will for the most part depend on the properties of the rewrite system itself.

Example 2.5.3 (The introductory example) We finish this chapter by summing up our solution to the non-commutation problem of Example 2.2.1. Recall that $C \equiv (\lambda y.\,[])x$ and $M \equiv xy$. In λc, the hole filling $C[M]$ is represented by $(\delta g.\,(\lambda y.\,g\,\langle y\rangle)\,x)\,\lceil \Lambda y.\,xy\rceil$. In λc, all rewrite sequences from this term end in the same term, as illustrated below.

$$\begin{array}{ccc}
(\delta g.\,(\lambda y.\,g\,\langle y\rangle)\,x)\,\lceil \Lambda y.\,xy\rceil & \to_\beta & (\delta g.\,g\,\langle x\rangle)\,\lceil \Lambda y.\,xy\rceil \\
\downarrow_{\mathit{fill}} & & \downarrow_{\mathit{fill}} \\
(\lambda y.\,(\Lambda y.\,xy)\,\langle y\rangle)\,x & \to_\beta & (\Lambda y.\,xy)\,\langle x\rangle \\
\downarrow_{\textcircled{\scriptsize m}\beta} & & \downarrow_{\textcircled{\scriptsize m}\beta} \\
(\lambda y.\,xy)x & \to_\beta & xx
\end{array}$$

Chapter 3

The context calculus λc

The context calculus λc is designed as a framework for lambda calculi with contexts. This chapter contains the definitions of the basic notions of the context calculus λc and the main properties of the framework. This chapter is concerned only with the untyped version of the framework; the succeeding chapters contain examples of syntactically typed applications.

This chapter is structured as follows.

In Section 3.1 the definition of the context calculus λc will be given. As we have explained in the previous chapter, in addition to the lambda-calculus constructors, in λc there are two more pairs of abstractors and applicators, namely $(\Lambda, \langle\,\rangle)$ and $(\delta, \lceil\rceil)$, and, moreover, a composition constructor $\circ$. The pair $(\Lambda, \langle\rangle)$ and the rewrite rule $(\textcircled{m}\beta)$ will be used for representing and computing communication. The pair $(\delta, \lceil\,\rceil)$ and the rewrite rule (*fill*) will be used for representing contexts and hole filling. The constructor $\circ$ and the rewrite rule $(\circ)$ will be used for composition. Hence, these added constructors with the rewrite rules together form the part of the calculus that will be concerned with representing and computing context-related operations.

In Section 3.2 two properties of rewriting in λc will be proved: the confluence property and the commutation property of the computations generated by pairs of possibly different rewrite relations. An experienced term rewriter can immediately see that λc has these two properties; here we work out the proofs. The confluence property gives the freedom to perform computation steps in an arbitrary order, and moreover, it guarantees the uniqueness of the result (if it exists) for any two coinitial computations. The commutation property is more specific about the independence of the order of computations, by saying that the computations of one rewrite relation commute with computations of any other rewrite relation. In particular, this property entails that the lambda calculus computations and context-related computations can be performed independently of each other. In other words, by the commutation property λc can be seen as a rewrite system of two independent levels: the level of lambda calculus and the level of the context-related machinery.

Both properties are proved via higher-order pattern rewrite systems, which are a

framework for term rewrite systems with binders. Since λc is a term rewrite system with binders, it can very naturally be written as a pattern rewrite system. The desired properties follow from the properties of pattern rewrite systems. Technically, the commutation property implies the confluence property, since the latter can be restated as the self-commutation property of the union of rewrite rules in λc. Therefore, first the proof of the commutation property of the rewrite rules will be presented in Section 3.2.4 and then the confluence proof in Section 3.2.5.

Section 3.3 inquires into the normalisation properties of rewriting. These include the strong and weak normalisation properties, that is, the properties whether each or some rewrite sequences lead to a result, respectively. In addition to these properties, the property of preservation of strong normalisation is discussed. As it turns out, the context calculus does not have weak and strong normalisation properties, but it does have the property of preservation of strong normalisation with respect to the untyped lambda calculus. However, the absence of weak or strong normalisation in the untyped framework is not bad. These normalisation properties are disturbed by some 'junk' that is present in the untyped framework. This 'junk' will be filtered out by syntactic typings, and we will return to these normalisation properties in each particular syntactic type system.

The importance of the two-level view on λc is that it indicates that the context calculus can be parametrised over an arbitrary rewrite system, instead of being defined as an extension of lambda calculus. That is, the term constructors Λ, $\langle\,\rangle$, δ, $\lceil\,\rceil$ and $\circ$ together with the rewrite rules ($\textcircled{m}\beta$), (*fill*) and ($\circ$) can be used to deal with contexts in an arbitrary term rewrite system with binders. Such a generalised context calculus will be considered in Section 3.4.

In Section 3.5 the work related to the techniques employed in the context calculus is discussed. First, the double role of higher-order rewriting in λc will be described: one in the proof of the properties mentioned above, and the other in the way communication between (representations of) λ-terms and contexts is established in λc. Second, the context calculus, as a system that captures the substitutions that arise from rewriting in a context, is compared to the calculi with explicit substitutions.

3.1 Definition of the context calculus λc

This section deals with the definitions of terms, substitution and rewrite rules of λc.

Let $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ as usual, and let $\mathcal{V}$ be a countably infinite set of variables.

Definition 3.1.1 (λc-terms) The terms of λc-calculus, or λc-terms for short, are inductively defined by

$$\begin{array}{rcrrr} U & ::= & u \mid & (\lambda u.U) \mid & (UU) \mid \\ & & (\Lambda_n u_1, \ldots, u_n.U) \mid & (U\langle U, \ldots, U\rangle_n) \mid & \\ & & (\delta_n u_1, \ldots, u_n.U) \mid & (U\lceil U, \ldots, U\rceil_n) \mid & (\circ_n(U, U, \ldots, U)) \end{array}$$

where $u, u_1, \ldots, u_n \in \mathcal{V}$ and $U, \ldots, U$ abbreviate n U's.

The set of λc-terms is denoted by $\textsc{Ter}(\lambda c)$.

Notation. The variables $u_1, \ldots, u_n$ in $(\Lambda_n u_1, \ldots, u_n.U)$ and $(\delta_n u_1, \ldots, u_n.U)$ will be abbreviated by $\vec{u}$, and terms $U_1, \ldots, U_n$ in the expressions $(U\langle U_1, \ldots, U_n\rangle_n)$, $(U\lceil U_1, \ldots, U_n\rceil_n)$ and $(\circ_n(U, U_1, \ldots, U_n))$ will be abbreviated by $\vec{U}$. In the multiple abstraction Λ_n, if $n = 0$ then we write $\Lambda\epsilon.U$. Note that the arity of $\langle\,\rangle_n$, $\lceil\,\rceil_n$ and $\circ_n$ is $n+1$. We will omit the index n and assume that the arities of Λ, $\langle\,\rangle$, δ, $\lceil\,\rceil$ and $\circ$ and the number of their arguments match. Furthermore, $\circ$ will be used in 'mix-fix' notation: $\circ_n(U, U, \ldots, U)$ will be denoted by $U \circ_n \vec{U}$. As usual, standard abbreviations regarding brackets apply, including association to the left. Also, as in lambda calculus, consecutive abstractions $\lambda u_1. \ldots \lambda u_n.U$ will be abbreviated by $\lambda\vec{u}.U$. However, consecutive Λ- or δ-abstractions are not abbreviated. In the remainder, the following convention considering typical elements will be maintained (if not explicitly stated otherwise): $i, j, m, n \in \mathbb{N}$, $u, u', u_i, v, w, \ldots \in \mathcal{V}$ and $U, U', U_i, V, W \ldots \in \textsc{Ter}(\lambda c)$.

The constructors λ and $\cdot$ (implicit applicator) are the λ-calculus constructors. The constructors $\Lambda_n \_ \ldots \_$ and $\delta_n \_ \ldots \_$ are multiple abstractors, which bind n variables simultaneously. The constructors $\_\langle \_, \ldots, \_\rangle_n$, $\_\lceil \_, \ldots, \_\rceil_n$ and $\_\circ_n(\_, \ldots, \_)$ are $n+1$-ary function symbols.

Example 3.1.2 A few examples of λc-terms are given. Some of these λc-terms can be interpreted as representations of objects in lambda calculus, using the representation methods described formally in Section 4.1. The last three λc-terms correspond to nothing in lambda calculus.

- $x\,y$
  (a representation of the λ-term $M \equiv x\,y$)
- $\delta h.\,(\lambda y.\,h\langle y\rangle)\,x$
  (a representation of the λ-context $C \equiv (\lambda y.\,[])\,x$)
- $\delta k_1, k_2.\,(k_1\langle\rangle)\,(\lambda z.\,k_2\langle z\rangle)$
  (a representation of the λ-context $D \equiv [\,](\lambda z.\,[])$)
- $\delta k_1.\,\delta k_2.\,(k_1\langle\rangle)\,(\lambda z.\,k_2\langle z\rangle)$
  (an alternative representation of the λ-context D with sequential hole abstraction)
- $(\delta h.\,(\lambda y.\,h\langle y\rangle)\,x)\lceil\Lambda y.\,x\,y\rceil$
  (a representation of the λ-context C to be filled with the λ-term M)
- $(\delta h.\,(\lambda y.\,h\langle y\rangle)\,x) \circ (\Lambda y.\,\delta k_1, k_2.\,(k_1\langle y\rangle)\,(\lambda z.\,k_2\langle y z\rangle))$
  (a representation of the composition of the λ-contexts C and D, with the binary $\circ$ in infix notation)
- $(\Lambda x.\,x) \circ y$
- $(\delta h.\,h\langle x\rangle)\lceil\Lambda x, y.\,x\rceil$

- $(\delta h.\, h\lceil h\rceil)\lceil\delta h.\, h\lceil h\rceil\rceil$.

As a formalism, the context calculus λc is a term rewrite system with binders. Hence, the free and bound variables in a λc-term are defined like in any other rewrite system with binders. Note that, while in lambda calculus α-conversion is not defined on meta-contexts, in λc α-conversion may safely be defined on the representations of lambda calculus meta-contexts within λc, because such representations are just λc-terms.

Like in the case of any other rewrite system with binders, λc-terms are considered equal up to α-conversion. Moreover, we assume that bound variables are renamed whenever necessary.

In λc, we need multiple substitutions, which are a straightforward pointwise extension of (single) substitutions.

Definition 3.1.3 (Substitution) For $U, \vec{V} \in \textsc{Ter}(\lambda c)$ and m distinct variables $\vec{v}$, where m is also the number of terms in $\vec{V}$, the result $U[\![\vec{v} := \vec{V}]\!]$ of substituting V_i for free occurrences of v_i in U $(1 \leq i \leq m)$ is defined as:

$$
\begin{array}{lcl}
u[\![\vec{v} := \vec{V}]\!] & = & \left\{ \begin{array}{lcl} V_i & : & \text{if } u = v_i \text{ for some } 1 \leq i \leq m \\ u & : & \text{otherwise} \end{array} \right. \\
(\lambda u.\, U')[\![\vec{v} := \vec{V}]\!] & = & \lambda u.\, (U'[\![\vec{v} := \vec{V}]\!]) \\
(U_1\, U_2)[\![\vec{v} := \vec{V}]\!] & = & (U_1[\![\vec{v} := \vec{V}]\!])\, (U_2[\![\vec{v} := \vec{V}]\!]) \\
(\Lambda \vec{u}.\, U')[\![\vec{v} := \vec{V}]\!] & = & \Lambda \vec{u}.\, (U'[\![\vec{v} := \vec{V}]\!]) \\
(U'\langle \vec{U} \rangle)[\![\vec{v} := \vec{V}]\!] & = & (U'[\![\vec{v} := \vec{V}]\!])\, \langle \vec{U}[\![\vec{v} := \vec{V}]\!] \rangle \\
(\delta \vec{u}.\, U')[\![\vec{v} := \vec{V}]\!] & = & \delta \vec{u}.\, (U'[\![\vec{v} := \vec{V}]\!]) \\
(U'\lceil \vec{U} \rceil)[\![\vec{v} := \vec{V}]\!] & = & (U'[\![\vec{v} := \vec{V}]\!])\, \lceil \vec{U}[\![\vec{v} := \vec{V}]\!] \rceil \\
(U' \circ \vec{U})[\![\vec{v} := \vec{V}]\!] & = & (U'[\![\vec{v} := \vec{V}]\!]) \circ (\vec{U}[\![\vec{v} := \vec{V}]\!])
\end{array}
$$

where $\vec{U}[\![\vec{v} := \vec{V}]\!]$ is an abbreviation for $(U_1[\![\vec{v} := \vec{V}]\!])$, …, $(U_n[\![\vec{v} := \vec{V}]\!])$. It is assumed that the bound variables u and $\vec{u}$ are renamed to avoid (unintended) variable capturing.

Although the context calculus is designed to work with λ-contexts as first-class objects, we still make use of a notion of meta-context over λc-terms (cf. Definition 2.1.3).

Definition 3.1.4 (Meta-contexts)

i) A meta-context in λc is a λc-term with some holes, denoted by [], all of which are considered different.

ii) Let C be a meta-context with n holes and let $\vec{U}$ be n λc-terms. Then the (meta-)hole filling results in the λc-term $C[\vec{U}]$ where the i^{th} hole of C has been replaced by U_i, for $1 \leq i \leq n$. After the (meta-)hole filling, variable capturing may occur: some free variables of $\vec{U}$ may become bound by the binders of C.

iii) Let C be a meta-context with n holes and let $D_1, \ldots, D_n$ be n meta-contexts where D_i has k_i holes, for $1 \leq i \leq n$. Then the (meta-)composition results in the meta-context $C[D_1, \ldots, D_n]$ where the i^{th} hole of C has been replaced by D_i, for $1 \leq i \leq n$. The meta-context $C[D_1, \ldots, D_n]$ contains $\sum_{1 \leq i \leq n} k_i$ holes. After the (meta-)composition, variable capturing may occur.

iv) Substitution, α-conversion and β-reduction are not defined on meta-contexts.

Computation in λc is defined on two levels: the lambda calculus level and the context-related level. Accordingly, the rewrite rule schemas (rewrite rules, for short), which generate computation in λc, are split into two collections.

Definition 3.1.5 (Context calculus λc) The context calculus λc is defined on terms of $\text{TER}(\lambda c)$ with rewrite relations induced by two collections of rewrite rules, the lambda calculus rewrite rules and the context rewrite rules. The two collections of rewrite rules are given below.

i) The lambda calculus rewrite rule is:

$$(\lambda u.U)V \;\to\; U[\![u := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$\begin{array}{ll}
(\Lambda\vec{u}.U)\langle\vec{V}\rangle \;\to\; U[\![\vec{u} := \vec{V}]\!] & (\textcircled{m}\beta) \\
(\delta\vec{u}.U)\lceil\vec{V}\rceil \;\to\; U[\![\vec{u} := \vec{V}]\!] & (\mathit{fill}) \\
(\delta_n\vec{u}.U)\circ_n(\Lambda\vec{v}_1.\delta\vec{v}'_1.V_1, \ldots, \Lambda\vec{v}_n.\delta\vec{v}'_n.V_n) & \\
\to\; \delta\vec{v}'_1, \ldots, \vec{v}'_n.U[\![u_1 := \Lambda\vec{v}_1.V_1, \ldots, u_n := \Lambda\vec{v}_n.V_n]\!]. & (\circ)
\end{array}$$

As usual, it is assumed that the bound variables in the rewrite rules are renamed to avoid variable capturing.

Notation. The rewrite relation generated by the context rewrite rules will be denoted by $\to_c$.

Remark 3.1.6 A reader who is familiar with combinatory reduction systems (CRSs, see [Klo80]) or higher-order rewrite systems (HRSs, see [MN98]) can see that λc can easily be written as a CRS or a HRS. Moreover, he/she can also see that λc is an orthogonal rewrite system. Consequently, he/she can conclude that the context calculus λc is confluent and that each pair of rewrite relations commute with each other.

Definition 3.1.7 (Underlying ARS of λc) The set of terms and the rewrite relations of λc define the ARS

$$\mathcal{A} = \langle \text{TER}(\lambda c), \to_\beta, \to_{\textcircled{m}\beta}, \to_{\mathit{fill}}, \to_\circ \rangle.$$

We will call this ARS the underlying ARS of λc.

We comment on the rewrite rules of λc. The lambda calculus rewrite rule is of course the rewrite rule (β) of lambda calculus. The context rewrite rules implement the context-related operations in λc: communication $(\textcircled{m}\beta)$, hole filling (fill) and composition $(\circ)$. The rewrite rules $(\textcircled{m}\beta)$, and (fill) denote actually one rewrite rule for each index $n \in \mathbb{N}$ of the abstraction and application. The rewrite rule $(\circ)$ denotes a rewrite rule for each combination of the indices involved, with only one condition: the indices of $\circ$ and of the first δ have to match. This condition is natural if one keeps in mind that a composition of λ-contexts, which is represented by the left-hand side of this rewrite rule, involves an outer context with n holes and n contexts to be filled into the holes of the outer context.

Example 3.1.8 Examples of the rewrite rules $(\textcircled{m}\beta)$, (fill) and $(\circ)$ are (let x, x', y be term variables, and other variables be hole variables):

i) $(\Lambda x, x'.\, U)\langle V, V'\rangle \;\to_{\textcircled{m}\beta}\; U[\![x := V, x' := V']\!]$

ii) $(\delta\epsilon.\, U)\lceil\,\rceil \;\to_{\mathit{fill}}\; U$

iii) $(\delta g.\, U) \circ (\Lambda x, x'.\, \delta h.\, V) \;\to_{\circ}\; \delta h.\, U[\![g := \Lambda x, x'.\, V]\!]$

iv) $(\delta g.\, U) \circ (\Lambda y.\, \delta h_1, h_2.\, V) \;\to_{\circ}\; \delta h_1, h_2.\, U[\![g := \Lambda y.\, V]\!]$

v) $(\delta g_1, g_2.\, U) \circ (\Lambda x.\, \delta h.\, V,\; \Lambda y.\, \delta k_1, k_2.\, W)$
$\to_{\circ}\; \delta h, k_1, k_2.\, U[\![g_1 := \Lambda x.\, V, g_2 := \Lambda y.\, W]\!].$

The last example illustrates the composition of a two-hole context with two contexts, where the hole abstractions of the latter contexts are shifted to the beginning of the resulting context. An example of a rewrite sequence in λc is

$$\begin{array}{ll} & (\delta h.\, \underline{(\lambda y.\, h\langle y\rangle)\, x}) \circ (\Lambda y.\, \delta k_1, k_2.\, (k_1\langle y\rangle)\,(\lambda z.\, k_2\langle y, z\rangle)) \\ \to_{\beta} & \underline{(\delta h.\, h\langle x\rangle) \circ (\Lambda y.\, \delta k_1, k_2.\, (k_1\langle y\rangle)\,(\lambda z.\, k_2\langle y, z\rangle))} \\ \to_{\circ} & \delta k_1, k_2.\, \underline{((\Lambda y.\, (k_1\langle y\rangle)\,(\lambda z.\, k_2\langle y, z\rangle))\,\langle x\rangle)} \\ \to_{\textcircled{m}\beta} & \delta k_1, k_2.\, (k_1\langle x\rangle)\,(\lambda z.\, k_2\langle x, z\rangle). \end{array}$$

Examples of λc-terms which are not redexes include:

- $(\Lambda_2 x, y.\, x)\langle x\rangle_1$, because the indices of Λ_2 and $\langle\,\rangle_1$ do not match, and
- $(\delta_2 h_1, h_2.\, h_1\langle x\rangle) \circ (\Lambda x, y.\, \delta k.\, y)$, because the indices of $\circ_1$ (implicit) and δ_2 do not match.

Remark 3.1.9 There is an analogy between the definition of composition of two (representations of) functions in lambda calculus and the definition of composition of two (representations of) contexts in λc. The analogy is based on the fact that within λc, lambda calculus contexts are represented as functions over hole variables. We will first consider unary functions and contexts with one hole; later we will indicate how the analogy can be applied for functions of arbitrary arity and contexts with many holes.

In lambda calculus, if f and g are two unary functions, then their composition is defined as (cf. Definition 6.2.8. in [Bar84])

"$f \circ g = \lambda x.\, f\,(g\,x)$."

Then, if $f \equiv \lambda y.\, M$ and $g \equiv \lambda z.\, N$ with $z \notin \mathrm{FVAR}(M)$,

$$\begin{array}{lcl} \text{“}f \circ g & = & \lambda x.\, f\,(g\,x)\text{”} \\ & \equiv & (\lambda y.\, M) \circ (\lambda z.\, N) \\ & = & \lambda x.\, (\lambda y.\, M)\,((\lambda z.\, N)\,x) \\ & = & \lambda x.\, (\lambda y.\, M)\,(N[\![z := x]\!]) \\ & = & \lambda z.\, (\lambda y.\, M)\,N \\ & = & \lambda z.\, M[\![y := N]\!]. \end{array}$$

Let now F and G be representations of contexts with one hole within λc. If the composition of F and G is to be formed, recall that G should have a communication preamble in order to handle communication. Thus, $F \equiv \delta h.\, U$ and $G \equiv \Lambda \vec{x}.\, \delta k.\, V$. Suppose without loss of generality that $k \notin \mathrm{FVAR}(U)$. In analogy to the definition of composition in lambda calculus, the first attempt to define the composition of F and G would be

"$F \circ G = \delta g.\, F\,\lceil G\,\lceil g \rceil\,\rceil$."

However, because of the communication preamble of G, some shifting of binders is necessary. The composition of F and G is computed as follows:

$$\begin{array}{lcl} \text{“}F \circ G & = & \delta g.\, F\,\lceil \Lambda \vec{y}.\, (G\,\langle \vec{y} \rangle)\,\lceil g \rceil\,\rceil\text{”} \\ & \equiv & \delta g.\, (\delta h.\, U)\,\lceil \Lambda \vec{y}.\, ((\Lambda \vec{x}.\, \delta k.\, V)\,\langle \vec{y} \rangle)\,\lceil g \rceil\,\rceil \\ & = & \delta g.\, (\delta h.\, U)\,\lceil \Lambda \vec{y}.\, (\delta k.\, V[\![\vec{x} := \vec{y}]\!])\,\lceil g \rceil\,\rceil \\ & = & \delta g.\, (\delta h.\, U)\,\lceil \Lambda \vec{y}.\, V\,[\![\vec{x} := \vec{y}]\!]\,[\![k := g]\!]\,\rceil \\ & = & \delta g.\, (\delta h.\, U)\,\lceil \Lambda \vec{x}.\, V[\![k := g]\!]\,\rceil \\ & = & \delta k.\, (\delta h.\, U)\,\lceil \Lambda \vec{x}.\, V\,\rceil \\ & = & \delta k.\, U[\![h := \Lambda \vec{x}.\, V]\!]. \end{array}$$

So, $(\delta h.\, U) \circ (\Lambda \vec{x}.\, \delta k.\, V) = \delta k.\, U[\![h := \Lambda \vec{x}.\, V]\!]$ and this is precisely what the composition rule for unary contexts implements. If the contexts are functions over many hole variables then an analogy can be given between the composition between an n-any function f and m_i-ary functions g_i for $1 \leq i \leq n$ (let $|\vec{x}_i| = m_i$)

"$f \circ (g_1, \ldots, g_n) = \lambda \vec{x}_1, \ldots, \vec{x}_n.\, f\,(g_1\,\vec{x}_1), \ldots, (g_n\,\vec{x}_n)$"

and the composition of contexts $F \equiv \delta \vec{h}.\, U$ and $G_1 \equiv \Lambda \vec{x}_1.\, \delta \vec{k}_1.\, V_1$ through $G_n \equiv \Lambda \vec{x}_n.\, \delta \vec{k}_n.\, V_n$:

"$F \circ \vec{G} = \delta \vec{g}_1, \ldots, \vec{g}_n.\, F\,\lceil \Lambda \vec{y}_1.\, (G_1\,\langle \vec{y}_1 \rangle)\,\lceil \vec{g}_1 \rceil, \ldots, \Lambda \vec{y}_n.\, (G_n\,\langle \vec{y}_n \rangle)\,\lceil \vec{g}_n \rceil\,\rceil$."

Remark 3.1.10 The set of terms of λc contains some 'junk', that is, terms that are not representations of objects of lambda calculus possibly involving context-related operations. In such superfluous λc-terms the number and the kind of arguments in the context machinery do not match. Consider for example the λc-terms

$(\Lambda x, y.\, x)\langle x\rangle$, $(\Lambda x.\, x) \circ y$ and $(\delta h.\, h\langle x\rangle)\lceil \Lambda x, y.\, x\rceil$. The term $(\Lambda x, y.\, x)\langle x\rangle$ is meaningless in lambda calculus because the communication it should represent does not match. The term $(\Lambda x.\, x) \circ y$ is meaningless because the form of the left argument of $\circ$ does not have the form of a representation of a context nor can it assume such a form by any substitution. The term $(\delta h.\, h\langle x\rangle)\lceil \Lambda x, y.\, x\rceil$ is meaningless too, because it reduces to the first term, which is meaningless. Such superfluous terms will be filtered out by syntactic typing in Chapters 4 and 5.

Remark 3.1.11 The context calculus can be defined more efficiently and, ultimately, translated to lambda calculus. The pairs of constructors $(\Lambda, \langle\,\rangle)$ and $(\delta, \lceil\,\rceil)$ with the corresponding rewrite rules have the same behaviour. Also, since compositions are functions on contexts, they can be defined as λc-terms. In the case of Example 2.5.2 in Chapter 2, let $\underline{comp} \equiv \lambda c.\, \delta d'.\, \delta h.\, c\lceil \Lambda x.\, (d'\langle x\rangle)\lceil h\rceil\rceil$. Furthermore, by encoding the single abstraction and single application as special cases of the corresponding multiple one, the only constructors that are really needed are Λ and $\langle\,\rangle$ and the rewrite rule $(\textcircled{m}\beta)$. These constructors and the rewrite rule (hence, all constructors and rewrite rules) can in turn be translated to the lambda calculus (using currying). This implies that the context calculus can be simulated within the lambda calculus where the computing of hole filling, composition and communication is performed in an algorithmic way, by rewriting groups of redexes simultaneously. However, we believe λc puts us at a suitable level of abstraction for working with contexts.

3.2 Commutation and confluence in λc

In this section the context calculus is proved to have two desirable properties: the commutation property of any pair of rewrite relations and the confluence property. Both proofs are conducted via pattern rewrite systems (PRSs), which are one of the formats of higher-order rewriting. Pattern rewrite systems offer a framework in which term rewrite systems with binders can uniformly be represented. A significant amount of theory has been developed for higher-order rewriting, from which we gratefully profit here. We choose for the format of pattern rewrite systems because the results we need are given in this format (cf. Prism theorem in [Oos95]).

In this section three systems will be considered: the context calculus λc, the PRS $\mathcal{H}$ and the system $\mathcal{H}_{\lambda c}$, which is a subsystem of $\mathcal{H}$ that corresponds to λc. The relationship between $\mathcal{H}$, $\mathcal{H}_{\lambda c}$ and λc is depicted in Figure 3.1. This section is organised as follows. In Section 3.2.1, the pattern rewrite system $\mathcal{H}$ is defined, by translating the constructors and the rewrite rules of the context calculus into the format of pattern rewrite systems. The pattern rewrite system $\mathcal{H}$ turns out to be orthogonal: there are no critical pairs and all rewrite rules are left-linear (see Section 1.3). In Section 3.2.2, $\mathcal{H}$ is restricted to a subsystem (sub-ARS, in the sense of Definition 1.1.13), called $\mathcal{H}_{\lambda c}$, which is closed under rewriting. Section 3.2.3 shows that there is a one–to–one correspondence between the terms and rewrite steps of $\mathcal{H}_{\lambda c}$ and the context calculus λc. In Section 3.2.4, from the properties of $\mathcal{H}$ and the

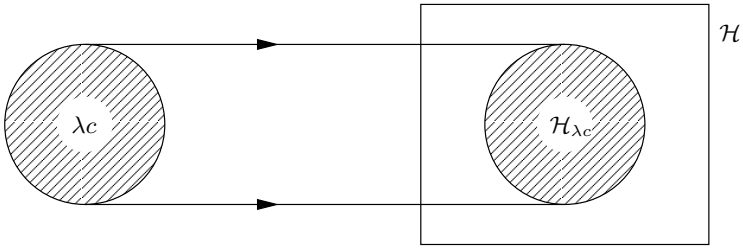


Figure 3.1: The context calculus λc in a higher-order format

theory of PRSs, it is shown that in $\mathcal{H}_{\lambda c}$ each pair of rewrite relations commutes, hence in the context calculus too. In Section 3.2.5, it is shown that λc is also confluent, because the commutation property entails confluence. In Section 3.5.1 we will also describe a second role of higher-order rewriting in λc: we will compare the higher-order rewriting technique for dealing with bindings with the way communication is established in λc.

Throughout this section we will briefly recall some definitions regarding PRSs. For an introduction to pattern rewrite systems and more pointers to relevant reading material, we refer to the preliminary Section 1.3.

Remark 3.2.1 The definition of $\mathcal{H}$ (3.2.3) is an example of how a term rewrite system with binders can be written in the format of PRS. The definition of the subset $\textsc{Ter}(\mathcal{H}_{\lambda c})$ (the alternative formulation in Proposition 3.2.9), which describes the elements of $\mathcal{H}$ that represent λc-terms, is also a standard definition and can be applied to the encoding of any term rewrite system. The definitions of both $\mathcal{H}$ and $\textsc{Ter}(\mathcal{H}_{\lambda c})$ are based on the encoding method presented by V. van Oostrom and F. van Raamsdonk in [OR93], where, in order to compare combinatory reduction systems (CRSs) of J.W. Klop with higher-order systems (HRSs) of T. Nipkow, an arbitrary CRS is translated into the HRS-format. Furthermore, the proof of closure of $\mathcal{H}_{\lambda c}$ under rewriting via a reformulation of the rewrite relation on the elements of $\mathcal{H}_{\lambda c}$ also has a standard form. These proofs can accordingly be adapted for a general case $\mathcal{H}_{\mathcal{A}}$ encoding an arbitrary term rewrite system $\mathcal{A}$.

Remark 3.2.2 At the beginning of this chapter we commented that the commutation property of the pairs of the rewrite relations of λc is a property by which λc can be split into two independent levels: the level of the lambda calculus and the level of context-related machinery. However, there is more to this property than just a nice result: this property will be essential in the chapters to come, where we will be working on subsystems of the framework λc. The notion of subsystem in these chapters is associated to the notion of indexed sub-ARS (cf. Definition 1.1.14). The notion of indexed sub-ARS is defined on ARSs with many rewrite relations, whereas the notion of standard sub-ARS is defined on ARSs with one rewrite relation. The notion of indexed sub-ARS allows a subsystem to include only a subset of rewrite relations of the supersystem. The difficulty with such a notion of subsystem is that

an indexed sub-ARS of a confluent system is not necessarily confluent, as it is the case in the standard notion of sub-ARS (see also Example 1.1.18). More specifically, given a pair of diverging rewrite sequences in a subsystem, the existence of a pair of converging rewrite sequences (in a supersystem) is not enough for confluence: it is also important that such a pair of converging rewrite sequences uses only the rewrite rules of the subsystem. The commutation property of the pairs of the rewrite relations of the framework λc ensures this is the case in any subsystem of λc. Thus, it pays off to prove the commutation property of the pairs of the rewrite relations in λc once, instead of proving confluence of each subsystem of λc.

3.2.1 The pattern rewrite system $\mathcal{H}$

A pattern rewrite system is a higher-order rewriting system with simply typed lambda calculus $\lambda^{\rightarrow}_{\overline{\eta}}$, with the rule $\overline{\eta}$, as the substitution calculus. In general, the set of types $\mathcal{T}$ of the calculus $\lambda^{\rightarrow}_{\overline{\eta}}$ is defined over a set of base types using the function constructor $\rightarrow$. Here, we restrict the set of base types to the singleton $\{0\}$. Furthermore, we assume there is a countably infinite set $\mathcal{V}^{\tau}$ of variables of type τ for each $\tau \in \mathcal{T}$, at our disposal. Let $\mathcal{V}^{\rightarrow}$ denote the union of typed variables, i.e. $\mathcal{V}^{\rightarrow} = \bigcup_{\tau\in\mathcal{T}} \mathcal{V}^{\tau}$. Without loss of generality, we will use for $\mathcal{V}^{0}$ the set of variables $\mathcal{V}$ of λc.

Notation. In order to avoid confusion between the symbol λ in $\lambda u.U$ of the context calculus, and the symbol λ in $\lambda u.\, s$ of the substitution calculus of PRSs during translations, the latter will be denoted only by the dot, for example $u.\, s$ instead of $\lambda u.\, s$ (such notation is customary in PRSs; see also notation in Section 1.3). A repeated abstraction $u.\, v.\, s$ will be abbreviated as $u, v.\, s$.

In general, as we have already mentioned in Chapter 1, the object language of pattern rewrite systems is generated from a set of typed function symbols (i.e. a signature) and a set of the rewrite rules. A rewrite rule is a pair of closed terms (i) : $l \rightarrow r$ of the same type. In a pattern rewrite system, the left-hand side l of each rewrite rule is a pattern. A pattern is a term of the form $\vec{z}.\, f(\vec{s})$ such that $f(\vec{s})$ is of a base type, and each z_i among $\vec{z}$ occurs free in $f(\vec{s})$ and has only ($\overline{\eta}$-normal forms of) pairwise distinct variables not among $\vec{z}$ as arguments.

In $\mathcal{H}$, the object language is based on λc: the preterms (i.e. expressions of $\mathcal{H}$) are generated from the function symbols corresponding to the constructors of the context calculus λc, and the rewrite rules are the pattern rewrite rules obtained by, loosely speaking, translating the rewrite rules of λc.

Definition 3.2.3 (Pattern rewrite system $\mathcal{H}$)

i) The signature $\mathcal{C}$ of $\mathcal{H}$ consists of the following function symbols ($n \in \mathbb{N}$):

$$
\begin{array}{lcll}
\mathsf{abs} & : & (0 \to 0) \to 0 & \\
\mathsf{app} & : & 0 \to 0 \to 0 & \\
\mathsf{mabs}_n & : & (\vec{0} \to 0) \to 0 & \text{with } |\vec{0}| = n \\
\mathsf{mapp}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1 \\
\mathsf{habs}_n & : & (\vec{0} \to 0) \to 0 & \text{with } |\vec{0}| = n \\
\mathsf{hf}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1 \\
\mathsf{comp}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1.
\end{array}
$$

ii) The set of rewrite rules $\mathcal{R}$ of $\mathcal{H}$ consists of the following rules:

$$z, z'.\, \mathsf{app}\ (\mathsf{abs}(u.\, zu))\ z' \ \to\ z, z'.\, zz' \tag{b}$$

$$z, \vec{z}.\, \mathsf{mapp}_n\ (\mathsf{mabs}_n(\vec{u}.\, z\vec{u}))\ \vec{z} \ \to\ z, \vec{z}.\, z\vec{z} \tag{mb}$$

$$z, \vec{z}.\, \mathsf{hf}_n\ (\mathsf{habs}_n(\vec{u}.\, z\vec{u}))\ \vec{z} \ \to_{\mathsf{fill}}\ z, \vec{z}.\, z\vec{z} \tag{fill}$$

$$
\begin{aligned}
&z, \vec{z}.\, \mathsf{comp}_n\ (\mathsf{habs}_n(\vec{u}.\, z\vec{u}))\ (\mathsf{mabs}(\vec{v}_1.\, \mathsf{habs}(\vec{v}\,'_1.\, z_1\vec{v}_1\vec{v}\,'_1)))\ldots \\
&\qquad\qquad (\mathsf{mabs}(\vec{v}_n.\, \mathsf{habs}(\vec{v}\,'_n.\, z_n\vec{v}_n\vec{v}\,'_n))) \\
&\to\ z, \vec{z}.\, \mathsf{habs}(\vec{v}\,'_1, \ldots, \vec{v}\,'_n.\, z(\mathsf{mabs}(\vec{v}_1.\, z_1\vec{v}_1\vec{v}\,'_1))\ldots(\mathsf{mabs}(\vec{v}_n.\, z_n\vec{v}_n\vec{v}\,'_n)))
\end{aligned} \tag{cmp}
$$

with $|\vec{z}| = n$ in (mb) and (fill).

iii) The PRS $\mathcal{H}$ is defined by $\mathcal{H} = (\mathcal{C}, \mathcal{R})$.

The PRS $\mathcal{H}$ is a second-order PRS, because all variables z are of type $\vec{0} \to 0$, which is of the order 2.

We recall some definitions from the preliminaries (Section 1.3). Let $[]^\tau$ be a special symbol for holes of type τ. A precontext C is a preterm with some holes in it, which can be filled simultaneously by possibly different preterms or precontexts. Terms are preterms in $\beta\overline{\eta}$-normal form. Contexts are precontexts in $\beta\overline{\eta}$-normal form. The rewrite relation is defined only on terms, and it is defined as $s \to_{\mathsf{i}} t$ if $s \twoheadleftarrow_\beta C[l]$ and $C[r] \twoheadrightarrow_\beta t$ where s and t are terms, (i) : $l \to r$ is a rewrite rule and C is a context. That is, a rewrite step consists of the extraction of the left-hand side of a rewrite rule, replacement by the corresponding right-hand side, and β-reduction.

Example 3.2.4 Let x, y, h, g, u be variables of type 0 and let z be a variable of type $0 \to 0$. Examples of terms of $\mathcal{H}$ are:

i) $\mathsf{app}\ x\ y$,

ii) $\mathsf{habs}(h.\, \mathsf{app}\ (\mathsf{abs}(y.\, \mathsf{mapp}\ h\ y))\ x)$,

iii) $\mathsf{comp}\ (\mathsf{habs}(h.\, \mathsf{app}\ (\mathsf{abs}(y.\, \mathsf{mapp}\ h\ y))\ x))$
$(\mathsf{mabs}(y.\, \mathsf{habs}(k_1, k_2.\, \mathsf{app}\ (\mathsf{mapp}\ k_1\ y)\ (\mathsf{abs}(z.\, \mathsf{mapp}\ k_2\ yz)))))$,

iv) $\mathsf{comp}_2\ x\ y$,

v) $\mathsf{abs}(u.\, zu)$, and

vi) $u.\,z$.

An example of a rewrite sequence in $\mathcal{H}$ is

$$\begin{array}{ll}
& \mathsf{comp}\ (\mathsf{habs}(h.\underline{\mathsf{app}\ (\mathsf{abs}(y.\,\mathsf{mapp}\ h\ y))\ x))} \\
& \quad (\mathsf{mabs}(y.\,\mathsf{habs}(k_1,k_2.\,\mathsf{app}\ (\mathsf{mapp}\ k_1\ y)\ (\mathsf{abs}(z.\,\mathsf{mapp}\ k_2\ yz))))) \\
\to_{\mathsf{b}} & \quad \underline{\mathsf{comp}\ (\mathsf{habs}(h.\,\mathsf{mapp}\ h\ x))} \\
& \quad \underline{(\mathsf{mabs}(y.\,\mathsf{habs}(k_1,k_2.\,\mathsf{app}\ (\mathsf{mapp}\ k_1\ y)\ (\mathsf{abs}(z.\,\mathsf{mapp}\ k_2\ yz)))))} \\
\to_{\mathsf{cmp}} & \quad \mathsf{habs}(k_1,k_2.\underline{\mathsf{mapp}\ (\mathsf{mabs}(y.\,\mathsf{app}\ (\mathsf{mapp}\ k_1\ y)\ (\mathsf{abs}(z.\,\mathsf{mapp}\ k_2\ y,z))))\ x)} \\
\to_{\mathsf{mb}} & \quad \mathsf{habs}(k_1,k_2.\,\mathsf{app}\ (\mathsf{mapp}\ k_1\ x)\ (\mathsf{abs}(z.\,\mathsf{mapp}\ k_2\ x,z))).
\end{array}$$

The notions of critical pair and of left-linearity can be lifted to higher-order rewriting (see Section 1.3). A critical pair is a tuple $(C[r\vec{s}], r'\vec{t})$ where $C[l\vec{s}] = l'\vec{t}$ is a most general overlap between two redexes of rewrite rules (i) : $l \to r$ and (i′) : $l' \to r'$. A rewrite rule is left-linear if each z_i occurs exactly once in the left-hand side $\vec{z}.\,f(\vec{s})$ of the rewrite rule. A pattern rewrite system is called orthogonal if there are no critical pairs and if all rewrite rules are left-linear.

Theorem 3.2.5 *$\mathcal{H}$ is orthogonal.*

Proof: By inspection of the rewrite rules of $\mathcal{H}$ we see that $\mathcal{H}$ is orthogonal. QED

3.2.2 The subsystem $\mathcal{H}_{\lambda c}$

From the context calculus point of view, $\mathcal{H}$ contains too many elements: as we have seen in the example above, $\mathcal{H}$ contains also terms like $\mathsf{comp}_2\ x\ y$, $\mathsf{abs}(u.\,zu)$ and $u.\,z$, which are intuitively meaningless in the context calculus. In this section, we will define the set of meaningful elements of $\mathcal{H}$ as $\textsc{Ter}(\mathcal{H}_{\lambda c})$, and give two descriptions of $\textsc{Ter}(\mathcal{H}_{\lambda c})$: one from the context calculus' point of view (Definition 3.2.6), and one from the point of view of higher-order rewriting (in Proposition 3.2.9). We will prove that $\mathcal{H}_{\lambda c} = \langle \textsc{Ter}(\mathcal{H}_{\lambda c}), \mathcal{R}_{\mathcal{H}} \rangle$ is a subsystem of $\mathcal{H}$, which is therefore closed under rewriting (Theorem 3.2.19). The crux of the proof is a reformulation of the rewrite relation (as explained in Intermezzo 3.2.10): from the definition of a rewrite step in $\mathcal{H}$, defined via terms and contexts that are in general not a part of the subsystem, to a formulation of a rewrite step in $\mathcal{H}_{\lambda c}$, via terms and contexts that are a part of the subsystem. The new formulation of a rewrite step is stated and proved to be equivalent to the old definition of the rewrite relation in Proposition 3.2.17.

The meaningful elements of $\mathcal{H}$, in the context calculus' viewpoint, are the elements which mimic the term formation of λc: starting from variables, elements are built using abstractors and functors (i.e. function symbols in $\mathcal{H}$) provided with the right number of (meaningful) arguments. The following definition describes such terms inductively.

Definition 3.2.6 ($\textsc{Ter}(\mathcal{H}_{\lambda c})$) Let $\textsc{Ter}(\mathcal{H}_{\lambda c})$ be the smallest subset of the pre-terms of $\mathcal{H}$ defined inductively as follows

i) variables of type 0 are elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, and

ii) if $u, \vec{u} \in \mathcal{V}^0$ and $s, t, \vec{s} \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ with $|\vec{u}| = |\vec{s}| = n$, then $\mathsf{abs}(u.\, s)$, $\mathsf{app}\ s\ t$, $\mathsf{mabs}_n(\vec{u}.\, s)$, $\mathsf{mapp}_n\ s\ \vec{s}$, $\mathsf{habs}_n(\vec{u}.\, s)$, $\mathsf{hf}_n\ s\ \vec{s}$, $\mathsf{comp}_n\ s\ \vec{s}$ are all elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$.

The elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ are indeed elements of $\mathcal{H}$ too, because they are well-typed preterms generated from the same set of types and constructors and from the restricted set of variables of $\mathcal{H}$ (only of type 0).

Let [] be a special symbol of type 0 for holes in $\mathcal{H}_{\lambda c}$. A (pre)context D over $\textsc{Ter}(\mathcal{H}_{\lambda c})$ is defined analogously to (pre)contexts over $\textsc{Ter}(\mathcal{H})$. Note that in $\mathcal{H}_{\lambda c}$ we consider only the (pre)contexts with holes of type 0, so the type annotation of a hole may be left implicit.

Notation. The indices of the function symbols will be left out, because, due to the well-typedness and the form of the elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, the indices have become superfluous: in $\mathsf{mabs}_n(\vec{u}.\, s)$ and $\mathsf{habs}_n(\vec{u}.\, s)$ the index n is coupled to the length of $\vec{u}$, while in $\mathsf{mapp}_n\ s\ \vec{s}$, $\mathsf{hf}_n\ s\ \vec{s}$ and $\mathsf{comp}_n\ s\ \vec{s}$ the index n is coupled to the length of $\vec{s}$.

From the point of view of $\mathcal{H}$, the elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ are preterms with computed substitutions (in β-normal form), in which all function symbols are provided with the right number of arguments (of type 0 and in $\overline{\eta}$-normal form), and with variables that stand only for another element of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ (i.e. no variables of type other than 0). This gives an alternative characterisation of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, which is stated in Proposition 3.2.9. For the proof of the proposition, two technical lemmas are needed.

Lemma 3.2.7 *Let s be a $\beta\overline{\eta}$-normal form. Then $s \equiv \lambda\vec{x}^{\vec{r}}.\, t_a\, \vec{t}$ where t_a is a constant or a variable and $\vec{t}$ are in $\beta\overline{\eta}$-normal form.*

Lemma 3.2.8 *Let s be a $\beta\overline{\eta}$-normal form. Then the subterms of s of a base type are in $\beta\overline{\eta}$-normal form.*

Proposition 3.2.9 *Let s be a preterm of $\mathcal{H}$. Then $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ if and only if s satisfies the following conditions*

($C1$) *s is in $\beta\overline{\eta}$-normal form,*

($C2$) *s is of type 0, and*

($C3$) *all variables in s (including the variables in the binders) are of type 0.*

Proof: In order to prove the 'only if'-part of the statement, one checks that $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ satisfies the conditions ($C1$), ($C2$) and ($C3$) by induction on the structure of s.

In order to prove the 'if'-part of the statement, one considers the preterms s of $\mathcal{H}$ that satisfy the conditions ($C1$)–($C3$). Such preterms s have a specific form,

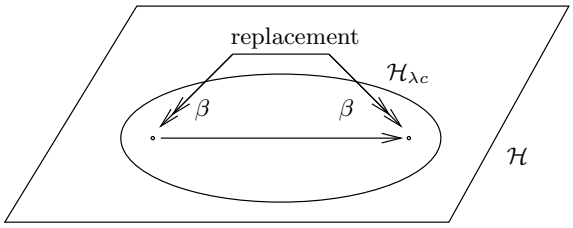


Figure 3.2: Rewrite steps of $\mathcal{H}_{\lambda c}$

which can be described as follows. Being in $\beta\overline{\eta}$-normal form (by $(C1)$), s is of the form $\vec{v}.\, a(\vec{t})$ where a is a variable or a function symbol and $\vec{t}$ are in $\beta\overline{\eta}$-normal form (by Lemma 3.2.7). Since s is of type 0 (by $(C2)$), the length of $\vec{v}$ is zero, implying that the preterm s must be of the form $a(\vec{t})$. Only the preterms of $\mathcal{H}$ of this form are considered.

Moreover, it can easily be verified that the subterms of type 0 of such an s (as a λ-term) satisfy the conditions $(C1)$ through $(C3)$ too (by using Lemma 3.2.8 for $(C1)$). This fact forms the basis of the inductive argument in the proof which proceeds by induction to the structure of such preterms. QED

Since the *terms* of $\mathcal{H}$ already satisfy the condition $(C1)$ by definition, this proposition can equivalently be restated as: a term s of $\mathcal{H}$ is an element of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ if and only if s satisfies $(C2)$ and $(C3)$. Hence, we will henceforth talk about terms instead of preterms.

In the remainder of this section we show that the set $\textsc{Ter}(\mathcal{H}_{\lambda c})$ is closed under rewriting in $\mathcal{H}$, that is, if $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ and $s \to t$ then $t \in \textsc{Ter}(\mathcal{H}_{\lambda c})$.

The difficulty of the proof of the closure lies in the fact that the rewrite steps of the subsystem $\mathcal{H}_{\lambda c}$ are inherited from $\mathcal{H}$, which are defined via terms and contexts of $\mathcal{H}$ that are not terms or contexts of the subsystem (see Figure 3.2). Recall that, if (i) : $l \to r$, is a rewrite rule, then $s \to_{\mathrm{i}} t$ is a rewrite step if and only if

$$s \twoheadleftarrow_\beta C[l] \text{ and } C[r] \twoheadrightarrow_\beta t.$$

Here, l and r are no terms of $\mathcal{H}_{\lambda c}$ and C is not a context over $\mathcal{H}_{\lambda c}$ (note that l, r and the hole are not of type 0). However, this definition can be reformulated closer to the subsystem, by keeping in mind that the subsystem is an encoding of the context calculus λc in the format of pattern rewrite systems. We will study the definition of rewrite steps in λc using λc-terms, to find a definition of rewrite steps in $\mathcal{H}_{\lambda c}$ using the encodings of λc-terms.

Intermezzo 3.2.10 Recall that an $\imath$-rewrite step in λc is defined by

$$C[L^*] \to_\imath C[R^*].$$

Here, $L \to R$ is the rewrite rule ($\imath$) of λc, C a context of λc, and L^* and R^* denote instances of L and R respectively. In these instances the meta-variables are replaced by λc-terms in such a way that variable capturing may occur.

In $\mathcal{H}_{\lambda c}$ this rewrite step will be mimicked by $s \to_{\mathsf{i}} t$ with

$$s \twoheadleftarrow_\beta D[l\vec{s}] \text{ and } D[r\vec{s}] \twoheadrightarrow_\beta t.$$

Here, s and t are encodings of $C[L^*]$ respectively $C[R^*]$, (i) : $l \to r$ is the rewrite rule of $\mathcal{H}_{\lambda c}$ associated with the rewrite rule ($\imath$) above, D is a context over $\mathcal{H}_{\lambda c}$ associated with the context C of λc. Moreover, the reductions $l\vec{s} \twoheadrightarrow_\beta s$ and $r\vec{s} \twoheadrightarrow_\beta t$ are 'internalised replacements' of the meta-variables in L^* and R^*. By this internalised replacement we mean a reduction which mimics the replacement of meta-variables together with the intended variable capturings. For example, the step in λc

$$\lambda y.\, (\Lambda x, x'.\, x)\, \langle y,\ (\lambda y'.\, y')\rangle \to_{(\mathbf{m})\beta} \lambda y.\, y$$

will be mimicked by the step in $\mathcal{H}_{\lambda c}$

$$\mathsf{abs}(y.\, \mathsf{mapp}\ (\mathsf{mabs}(x, x'.\, x))\ y\ (\mathsf{abs}(y'.\, y'))) \ \to_{\mathsf{mb}}\ \mathsf{abs}(y.\, y)$$

with

$$\mathsf{abs}(y.\, \mathsf{mapp}\ (\mathsf{mabs}(x, x'.\, x))\ y\ (\mathsf{abs}(y'.\, y'))) \twoheadleftarrow_\beta$$
$$\mathsf{abs}(y.\, [])[(z, z_1, z_2.\, \mathsf{mapp}\ (\mathsf{mabs}(u_1, u_2.\, z u_1 u_2))\ z_1 z_2)\ (x, x'.\, x)\ y\ (\mathsf{abs}(y'.\, y'))]$$

and

$$\mathsf{abs}(y.\, [])[(z, z_1, z_2.\, z z_1 z_2)\ (x, x'.\, x)\ y\ (\mathsf{abs}(y'.\, y'))] \twoheadrightarrow_\beta\ \mathsf{abs}(y.\, y)\ .$$

The correspondence between the rewrite steps in λc and $\mathcal{H}_{\lambda c}$ will be proved in the next section; here we are only concerned with the equivalence of the two definitions of rewrite steps originating from the elements of $\mathcal{H}_{\lambda c}$ and the closure of $\mathcal{H}_{\lambda c}$ under (the new definition of) rewriting.

Because the terms of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ have unusual subterm and superterm relations, we start with two technical propositions which specify the conditions for the closure of these two relations.

Proposition 3.2.11 *If $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ then all subterms of type 0 of s are terms of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ too.*

Proof: Each $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ satisfies the conditions $(C1)$ through $(C3)$, by Proposition 3.2.9. Considering such an s as a $\lambda\beta\overline{\eta}$-term, one verifies that its subterms of type 0 satisfy the conditions $(C1)$ through $(C3)$, too (by using Lemma 3.2.8 for $(C1)$). By Proposition 3.2.9 again, such subterms are elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$. QED

Proposition 3.2.12 *Let $\vec{s}$ be n terms of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ and let D be a context over $\textsc{Ter}(\mathcal{H})$ with n holes. Then, $D[\vec{s}] \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ if and only if D is a context over $\textsc{Ter}(\mathcal{H}_{\lambda c})$.*

Proof: Let D be a context over $\textsc{Ter}(\mathcal{H})$ with n holes and let $\vec{s} \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ with $|\vec{s}| = n$.

To prove the 'only if'-part, suppose $D[\vec{s}] \in \textsc{Ter}(\mathcal{H}_{\lambda c})$. In order for hole filling to be defined on D and $\vec{s}$, the type of s_i and of the i^{th} hole in D must be the same (for $1 \leq i \leq n$). Since all s_i are elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, the type of all s_i is 0 (by Proposition 3.2.9). Then the type of all holes in D must be 0, too. The rest of the proof proceeds by induction to D (a context over $\textsc{Ter}(\mathcal{H})$).

The 'if'-part is proved by induction to context D over $\textsc{Ter}(\mathcal{H}_{\lambda c})$. QED

If a rewrite step is initiated from a term of $\mathcal{H}_{\lambda c}$ of a base type, then an 'internalised replacement' of the meta-variables can be identified in the arguments of the left-hand and right-hand sides of the rewrite rule used. This is proved in Proposition 3.2.15. Note also that the reduct is of the same base type, because the rewrite relation of $\mathcal{H}$ preserves types.

We also need two technical lemmas in the substitution calculus.

Lemma 3.2.13 *Let s be a $\overline{\eta}$-normal form of some base type. If t_a is a variable or a constant of type $\vec{\tau} \to \mathtt{a}$ where $\mathtt{a}$ is a base type, and t_a is a subterm of s, then t_a is in a context of the form $t_a\vec{t}$ where $\vec{t} : \vec{\tau}$, also in $\overline{\eta}$-normal form.*

Lemma 3.2.14 *Let t be a $\beta\overline{\eta}$-normal form of some base type. Let x be a variable of the same base type. Then $s[\![x := t]\!]$ is in $\beta\overline{\eta}$-normal form if and only if s is in $\beta\overline{\eta}$-normal form.*

Proposition 3.2.15 *Let $t_1, t_2 \in \textsc{Ter}(\mathcal{H})$, both of type 0. Let (i) : $l \to r$ be a rewrite rule. If $t_1 \to_{\mathsf{i}} t_2$ then there are a context D over $\mathcal{H}$ and terms $\vec{s}$ of $\mathcal{H}$ such that $l\vec{s} : 0$, $t_1 \equiv D[(l\vec{s})\downarrow_\beta]$ and $t_2 \equiv D[(r\vec{s})\downarrow_\beta]$.*

Proof: Let $t_1, t_2 \in \textsc{Ter}(\mathcal{H})$, both of type 0, let (i) : $l \to r$ be a rewrite rule, and let $t_1 \to_{\mathsf{i}} t_2$. Then there is a context C over $\mathcal{H}$ such that $t_1 \twoheadleftarrow_\beta C[l]$ and $C[r] \twoheadrightarrow_\beta t_2$, per definition of rewrite steps in higher-order rewriting. Due to the preservation of types by β, and because t_1 and t_2 are both of type 0, so is C.

The types of l, r and the hole in C are the same. By inspection of the rewrite rules of $\mathcal{H}$, one sees that $l : \vec{\tau} \to 0$ for $\tau_i = \vec{0} \to 0$. Thus, $[] : \vec{\tau} \to 0$. Since C is in $\beta\overline{\eta}$-normal form, the hole is in a context of the form $[]\vec{s} : 0$ with $\vec{s} : \vec{\tau}$ and all $\vec{s}$ in $\beta\overline{\eta}$-normal form, by Lemma 3.2.13. Let D be the context such that $C \equiv D[[]\vec{s}]$. Note that D is in $\beta\overline{\eta}$-normal form, since the type of the hole in D is 0 (by Lemma 3.2.14).

Consider the rewrite sequences $D[l\vec{s}] \twoheadrightarrow_\beta t_1$ and $D[r\vec{s}] \twoheadrightarrow_\beta t_2$. Since D is in $\beta\overline{\eta}$-normal form and the type of $l\vec{s}$ and $r\vec{s}$ is 0, these two rewrite sequences consist of reducing $l\vec{s}$ and $r\vec{s}$ to their $\beta\overline{\eta}$-normal forms, respectively. Consequently, $t_1 \equiv D[(l\vec{s})\downarrow_\beta]$ and $t_2 \equiv D[(r\vec{s})\downarrow_\beta]$.

[This proof can be generalised to any base type.] QED

The terms of the replacement part are terms of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ with a prefix that handles variable bindings. This prefix takes care of the bindings that in λc occur by variable capturing which occurs during the replacement of the meta-variables.

Proposition 3.2.16 *Let* (i) : $l \to r$ *be a rewrite rule of* $\mathcal{H}$. *Let* $\vec{s}$ *be* n *terms of* $\mathcal{H}$. *If* $(l\vec{s})\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$ *then* $s_i \equiv \vec{x}_i.s'_i$ *with* $\vec{x}_i, s'_i \in \text{TER}(\mathcal{H}_{\lambda c})$ *for all* i *with* $1 \leq i \leq n$.

Proof: Let (i) : $l \to r$ be a rewrite rule of $\mathcal{H}$ and let $\vec{s}$ be n terms of $\mathcal{H}$. Suppose $(l\vec{s})\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$. By inspection of the rewrite rules of $\mathcal{H}$, note that $l \equiv \vec{z}.l' : \vec{\tau} \to 0$ with $z_i : \tau_i = \vec{0}_i \to 0$. Then it holds that $s_i : \vec{0}_i \to 0$ for $1 \leq i \leq |\vec{s}|$. Furthermore, since each s_i is in $\beta\overline{\eta}$-normal form, s_i is of the form $\vec{x}_i.s'_i$ where s'_i is in $\beta\overline{\eta}$-normal form and of type 0, and $\vec{x}_i : \vec{0}_i$. Using Proposition 3.2.9, $\vec{x}_i \in \text{TER}(\mathcal{H}_{\lambda c})$.

It remains to be checked that each s'_i contains only variables of type 0. Consider the reduction $l\vec{s} \twoheadrightarrow_\beta (l\vec{s})\downarrow_\beta$ and $l \equiv \vec{z}.l'$ with $z_i : \vec{0}_i \to 0$. Because simply typed λ-terms have unique β-normal forms (with β being confluent and strongly normalising), it holds that $((\vec{z}.l')\vec{s})\downarrow_\beta = (l'[\![\vec{z} := \vec{s}]\!])\downarrow_\beta$. Since variables other than $\vec{z}$ in l are of type 0, β-reduction does not introduce any new variables (up to renaming, which is type-preserving), and $(l'[\![\vec{z} := \vec{s}]\!])\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$, each s'_i must contain only variables of type 0. QED

The two definitions of rewrite steps in $\mathcal{H}$, originating from an element of $\mathcal{H}_{\lambda c}$ are equivalent.

Proposition 3.2.17 *Let* $s, t \in \text{TER}(\mathcal{H})$. *Let* (i) : $l \to r$ *be a rewrite rule of* $\mathcal{H}$. *Suppose* $s \in \text{TER}(\mathcal{H}_{\lambda c})$. *Then,* $s \to_{\mathsf{i}} t$ *if and only if there is a context* D *of* $\mathcal{H}_{\lambda c}$ *and* $\vec{s}$ *with* $l\vec{s} : 0$, $s_i \equiv \vec{x}_i.s'_i$ *and* $\vec{x}_i, s'_i \in \text{TER}(\mathcal{H}_{\lambda c})$ *for* $1 \leq i \leq |\vec{s}|$ *such that*

$$s \equiv D[(l\vec{s})\downarrow_\beta] \text{ and } t \equiv D[(r\vec{s})\downarrow_\beta].$$

Proof: To prove the 'only if'-part, suppose $s \to_{\mathsf{i}} t$. Since s is of a base type (being an element of $\text{TER}(\mathcal{H}_{\lambda c})$), there is a context D over $\mathcal{H}$ and terms $\vec{s}$ of $\mathcal{H}$ such that $l\vec{s} : 0$, $s \equiv D[(l\vec{s})\downarrow_\beta]$ and $t \equiv D[(r\vec{s})\downarrow_\beta]$, by Proposition 3.2.15. Since $(l\vec{s})\downarrow_\beta$ is a subterm of type 0 of s, it is an element of $\text{TER}(\mathcal{H}_{\lambda c})$ by Proposition 3.2.11. Then D is a context over $\text{TER}(\mathcal{H}_{\lambda c})$ by Proposition 3.2.12. Because $(l\vec{s})\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$, $s_i \equiv \vec{x}_i.s'_i$ and $\vec{x}_i, s'_i \in \text{TER}(\mathcal{H}_{\lambda c})$ for $1 \leq i \leq |\vec{s}|$, by Proposition 3.2.16.

To prove the 'if'-part, let D be a context of $\mathcal{H}_{\lambda c}$ and $\vec{s}$ with $l\vec{s} : 0$, $s_i \equiv \vec{x}_i.s'_i$ and $\vec{x}_i, s'_i \in \text{TER}(\mathcal{H}_{\lambda c})$ such that

$$s \equiv D[(l\vec{s})\downarrow_\beta] \text{ and } t \equiv D[(r\vec{s})\downarrow_\beta].$$

In that case, let $C \equiv D[[]\vec{s}]$. Then, $s \twoheadleftarrow C[l]$ and $C[r] \twoheadrightarrow t$, and hence, $s \to_{\mathsf{i}} t$. QED

The next two statements finish the proof of the closure by stating that the contractions originating from elements of $\mathcal{H}_{\lambda c}$ are within $\mathcal{H}_{\lambda c}$, and consequently, the rewrite steps originating from elements of $\mathcal{H}_{\lambda c}$, too.

Proposition 3.2.18 *Let* (i) : $l \to r$ *be a rewrite rule of* $\mathcal{H}$. *Let* $\vec{s}$ *be terms of* $\mathcal{H}$. *Then, if* $(l\vec{s})\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$ *then* $(r\vec{s})\downarrow_\beta \in \text{TER}(\mathcal{H}_{\lambda c})$.

Proof: Assume $(l\vec{s})\downarrow_\beta \in \mathcal{H}_{\lambda c}$. By Proposition 3.2.16, $s_i \equiv \vec{x}_i.s'_i$ with $\vec{x}_i, s'_i \in \text{TER}(\mathcal{H}_{\lambda c})$ for all i with $1 \leq i \leq |\vec{s}|$. Moreover, we know that the right-hand side

r is of the same type as l and in $\beta\overline{\eta}$-normal form, that is, $r \equiv \vec{z}.r'$ where r' is in $\beta\overline{\eta}$-normal form and of type 0. We also know that $(r\vec{s})\downarrow_\beta = (r'[\![\vec{z} := \vec{s}]\!])\downarrow_\beta$ because each term has a unique β-normal form.

Then $(r\vec{s})\downarrow_\beta$ satisfies the conditions $(C1)$–$(C3)$:

$(C1)$**:** Because $[\![\vec{z} := \vec{s}]\!]$ preserves $\overline{\eta}$-normal forms, (by Proposition 1.2.52), $r'[\![\vec{z} := \vec{s}]\!]$ is again in $\overline{\eta}$-normal form. Since β-reduction preserves $\overline{\eta}$-normal forms, (by Proposition 1.2.52(ii)), $(r'[\![\vec{z} := \vec{s}]\!])\downarrow_\beta$ is in $\beta\overline{\eta}$-normal form. That is, $(r\vec{s})\downarrow_\beta$ is in $\beta\overline{\eta}$-normal form.

$(C2)$**:** Since $l\vec{s}$ is of type 0, so is $r\vec{s}$. Since $\rightarrow_\beta$ is type-preserving, the type of $(r\vec{s})\downarrow_\beta$ is 0, too.

$(C3)$**:** The only variables in $r\vec{s}$ of type other than 0 are among z_i's. These variables will eventually be substituted by $\vec{s}$ in r' along the reduction $r\vec{s} \twoheadrightarrow_\beta r\vec{s}\downarrow_\beta$. Since by β-reduction no new variables can appear (although some variables may be duplicated or renamed in a type-preserving way), all variables in $(r'[\![\vec{z} := \vec{s}]\!])\downarrow_\beta$ are then of type 0. QED

Theorem 3.2.19 *Let $s \in \mathrm{TER}(\mathcal{H}_{\lambda c})$ and let* (i) *be a rewrite rule of $\mathcal{H}$. If $s \rightarrow_{\mathsf{i}} t$ then $t \in \mathrm{TER}(\mathcal{H}_{\lambda c})$.*

Proof: We prove this statement only for the case of the rewrite sequence of length 1; the statement follows by induction to the length of the reduction.

Suppose $s \in \mathrm{TER}(\mathcal{H}_{\lambda c})$, (i) : $l \rightarrow r$ and $s \rightarrow_{\mathsf{i}} t$ is a rewrite step in $\mathcal{H}$. Then there is a context D of $\mathcal{H}_{\lambda c}$ and $\vec{s}$ with $l\vec{s} : 0$, $s_i \equiv \vec{x}_i.s'_i$ and $\vec{x}_i, s'_i \in \mathrm{TER}(\mathcal{H}_{\lambda c})$ for $1 \leq i \leq |\vec{s}|$ such that

$$s \equiv D[(l\vec{s})\downarrow_\beta] \text{ and } t \equiv D[(r\vec{s})\downarrow_\beta].$$

Since $(l\vec{s})\downarrow_\beta$ is a subterm of type 0 of s, it is an element of $\mathrm{TER}(\mathcal{H}_{\lambda c})$ by Proposition 3.2.11. Then, by Proposition 3.2.18, $(r\vec{s})\downarrow_\beta$ is an element of $\mathrm{TER}(\mathcal{H}_{\lambda c})$. By Proposition 3.2.12, $D[(r\vec{s})\downarrow_\beta] \equiv t$ is also an element of $\mathrm{TER}(\mathcal{H}_{\lambda c})$. QED

Corollary 3.2.20 (Indexed sub-PRS $\mathcal{H}_{\lambda c}$)
The system $\mathcal{H}_{\lambda c} = \langle \mathrm{TER}(\mathcal{H}_{\lambda c}), \mathcal{R}_{\mathcal{H}} \rangle$ is an indexed sub-PRS of $\mathcal{H}$.

Proof: The underlying ARS of $\mathcal{H}_{\lambda c}$ satisfies the conditions of the definition of an indexed sub-ARS of the underlying ARS of $\mathcal{H}$:

i) The set of terms $\mathrm{TER}(\mathcal{H}_{\lambda c})$ is a subset of the set of terms of $\mathcal{H}$.

ii) The rewriting relations of $\mathcal{H}_{\lambda c}$ are the restrictions of the same rewrite relations of $\mathcal{H}$, because the rewriting relations are generated by the same rewrite rules $\mathcal{R}_{\mathcal{H}}$.

iii) By Theorem 3.2.19, the set of terms $\mathrm{TER}(\mathcal{H}_{\lambda c})$ is closed under each rewrite relation of $\mathcal{H}$. QED

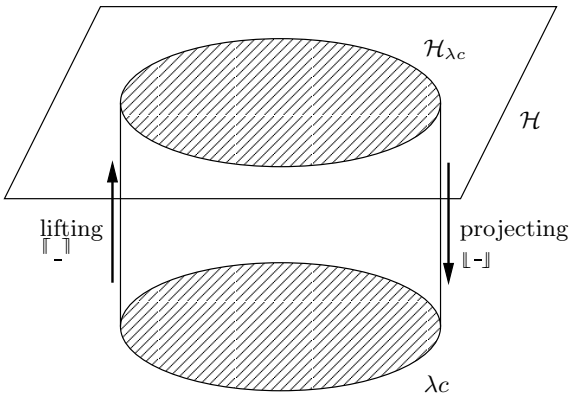


Figure 3.3: Lifting and projecting

Remark 3.2.21 Whenever it will be more convenient, we will use the new definition of rewrite steps, based on Proposition 3.2.17 instead of the usual definition in $\mathcal{H}$.

3.2.3 The correspondence between λc and $\mathcal{H}_{\lambda c}$

The statement of this section is 'λc is $\mathcal{H}_{\lambda c}$'. For the purpose of making the correspondence concrete, two translation functions are defined (see Figure 3.3): lifting (Definition 3.2.22), which translates λc-terms to $\mathcal{H}_{\lambda c}$; and projection (Definition 3.2.24), which translates the terms of $\mathcal{H}_{\lambda c}$ to λc-terms. The translations are each others inverse (Theorem 3.2.25), and as such they define a one–to–one correspondence between the terms of λc and $\mathcal{H}_{\lambda c}$. Moreover, the translations preserve the rewrite steps: if U reduces to V, then the lifting of U reduces to the lifting of V (Theorem 3.2.30); and if s reduces to t, then the projection of s reduces to the projection of t (Theorem 3.2.31). Such a correspondence implies that the properties of the rewrite relations will be the same in both systems.

The definition of lifting spells out the intuition used when defining $\mathcal{H}$ and $\mathcal{H}_{\lambda c}$.

Definition 3.2.22 (Lifting $\textsc{Ter}(\lambda c)$ to $\textsc{Ter}(\mathcal{H}_{\lambda c})$)

i) Let $U \in \textsc{Ter}(\lambda c)$. The lifting of U to $\textsc{Ter}(\mathcal{H}_{\lambda c})$, $\llbracket U \rrbracket$ is defined by the structural induction on U:

$$
\begin{array}{lcl}
\llbracket u \rrbracket & = & u \\
\llbracket\ \lambda u.U\ \rrbracket & = & \mathsf{abs}(u.\llbracket U \rrbracket) \\
\llbracket\ U\,V\ \rrbracket & = & \mathsf{app}\ \llbracket U \rrbracket\ \llbracket V \rrbracket \\
\llbracket\ \Lambda\vec{u}.U\ \rrbracket & = & \mathsf{mabs}(\vec{u}.\llbracket U \rrbracket) \\
\llbracket\ U\langle\vec{U}\rangle\ \rrbracket & = & \mathsf{mapp}\ \llbracket U \rrbracket\ \llbracket \vec{U} \rrbracket \\
\llbracket\ \delta\vec{u}.U\ \rrbracket & = & \mathsf{habs}(\vec{u}.\llbracket U \rrbracket) \\
\llbracket\ U\lceil\vec{U}\rceil\ \rrbracket & = & \mathsf{hf}\ \llbracket U \rrbracket\ \llbracket \vec{U} \rrbracket \\
\llbracket\ U \circ \vec{U}\ \rrbracket & = & \mathsf{comp}\ \llbracket U \rrbracket\ \llbracket \vec{U} \rrbracket
\end{array}
$$

where the variables u, $\vec{u}$ on the right-hand sides are all of type 0.

ii) Lifting is straightforwardly extended to contexts where $\llbracket\ []\ \rrbracket = []$ of type 0.

Notationally, we will make no distinction between (untyped) u of λc and (typed) u of $\lambda^{\rightarrow}$, but assume it is clear from the surrounding text which one is meant.

Lifting is a well-defined function, that is, the lifting of an element of λc results in a unique element of $\mathcal{H}_{\lambda c}$. Moreover, lifting commutes with the meta-operations of filling holes and substitution.

Proposition 3.2.23

i) *Let $\vec{U} \in \textsc{Ter}(\lambda c)$ and C be a context with n holes over λc-terms. Then*

$$\llbracket C[\vec{U}] \rrbracket = \llbracket C \rrbracket[\llbracket \vec{U} \rrbracket].$$

ii) *Let $U, \vec{V} \in \textsc{Ter}(\lambda c)$ and $\vec{v}$ be n distinct variables of $\textsc{Ter}(\lambda c)$. Then*

$$\llbracket U\,[\![\vec{v} := \vec{V}]\!]\ \rrbracket = \llbracket U \rrbracket [\![\llbracket \vec{v} \rrbracket := \llbracket \vec{V} \rrbracket]\!].$$

Proof: Note that the square brackets [ ] in $\llbracket C[\vec{U}] \rrbracket$ denote the meta-operation of hole filling in λc, whereas the same brackets in $\llbracket C \rrbracket[\llbracket \vec{U} \rrbracket]$ denote the meta-operation of hole filling in higher-order systems.

i) The proof is done by induction to the structure of the context C over λc-terms.

ii) First of all, note that $\llbracket [\![\vec{v}]\!] := \llbracket \vec{V} \rrbracket\ \rrbracket$ is a valid higher-order substitution since for all $1 \leq i \leq n$, $\llbracket v_i \rrbracket = v_i$ and $\llbracket V_i \rrbracket$ are of the same type (namely, 0) and $\llbracket V_i \rrbracket$ is in $\beta\overline{\eta}$-normal form, by Proposition 3.2.9 and well-definedness of lifting. The proof proceeds by induction to U. QED

The definition of projection of s to $\textsc{Ter}(\lambda c)$, $\lfloor\!\lfloor s \rfloor\!\rfloor$ is given by exchanging the left-hand and right-hand sides in the definition of lifting.

Definition 3.2.24 (Projection of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ to $\textsc{Ter}(\lambda c)$)

i) Let $s \in \text{TER}(\mathcal{H}_{\lambda c})$. The projection of s to $\text{TER}(\lambda c)$, $\llparenthesis s \rrparenthesis$ is defined by structural induction on s:

$$\begin{array}{lcl}
\llparenthesis u \rrparenthesis & = & u \\
\llparenthesis\ \mathsf{abs}(u.\, s)\ \rrparenthesis & = & \lambda u.\, \llparenthesis s \rrparenthesis \\
\llparenthesis\ \mathsf{app}\ s\ t\ \rrparenthesis & = & \llparenthesis s \rrparenthesis\, \llparenthesis t \rrparenthesis \\
\llparenthesis\ \mathsf{mabs}(\vec{u}.\, s)\ \rrparenthesis & = & \Lambda \vec{u}.\, \llparenthesis s \rrparenthesis \\
\llparenthesis\ \mathsf{mapp}\ s\ \vec{s}\ \rrparenthesis & = & \llparenthesis s \rrparenthesis\, \langle \llparenthesis \vec{s} \rrparenthesis \rangle \\
\llparenthesis\ \mathsf{habs}(\vec{u}.\, s)\ \rrparenthesis & = & \delta \vec{u}.\, \llparenthesis s \rrparenthesis \\
\llparenthesis\ \mathsf{hf}\ s\ \vec{t}\ \rrparenthesis & = & \llparenthesis s \rrparenthesis\, \lceil \llparenthesis \vec{t} \rrparenthesis \rceil \\
\llparenthesis\ \mathsf{comp}\ s\ \vec{s}\ \rrparenthesis & = & \llparenthesis s \rrparenthesis \circ \llparenthesis \vec{s} \rrparenthesis .
\end{array}$$

ii) Projection is straightforwardly extended to contexts where $\llparenthesis\ []\ \rrparenthesis = []$.

Projection is a well-defined function, that is, the projection of an element of $\mathcal{H}_{\lambda c}$ results in a unique element of λc.

The correspondence between the terms of λc and $\mathcal{H}_{\lambda c}$ is established by the next proposition, which states that lifting and projection are each others inverse, and as such, they define the one–to–one correspondence.

Theorem 3.2.25

i) Let $U \in \text{TER}(\lambda c)$. Then $\llparenthesis \ulcorner U \urcorner \rrparenthesis = U$.

ii) Let $s \in \text{TER}(\mathcal{H}_{\lambda c})$. Then $\ulcorner \llparenthesis s \rrparenthesis \urcorner = s$.

Proof: The proofs are conducted by structural induction on $U \in \text{TER}(\lambda c)$ and $s \in \text{TER}(\mathcal{H}_{\lambda c})$, respectively. QED

Projecting commutes with the (meta-level) hole-filling operation and substitution, as is shown in the next two propositions.

Proposition 3.2.26

i) Let $\vec{s} \in \text{TER}(\mathcal{H}_{\lambda c})$ and D be a context over the terms of $\text{TER}(\mathcal{H}_{\lambda c})$ with n holes. Then

$$\llparenthesis D[\vec{s}] \rrparenthesis = \llparenthesis D \rrparenthesis [\llparenthesis \vec{s} \rrparenthesis].$$

ii) Let $s, \vec{t} \in \text{TER}(\mathcal{H}_{\lambda c})$ and $\vec{v}$ be n distinct variables of type 0. Then

$$\llparenthesis s\, [\![\vec{v} := \vec{t}]\!]\ \rrparenthesis = \llparenthesis s \rrparenthesis [\![\llparenthesis \vec{v} \rrparenthesis := \llparenthesis \vec{t} \rrparenthesis]\!].$$

Proof:

i) The proof uses the fact that lifing and projection are each others inverse:

$$\begin{array}{lcll}
\llparenthesis D[\vec{s}] \rrparenthesis & = & \llparenthesis \ulcorner \llparenthesis D \rrparenthesis \urcorner [\ulcorner \llparenthesis \vec{s} \rrparenthesis \urcorner] \rrparenthesis & \text{[Theorem 3.2.25}(ii)\text{]} \\
& = & \llparenthesis \ulcorner \llparenthesis D \rrparenthesis\, [\llparenthesis \vec{s} \rrparenthesis] \urcorner \rrparenthesis & \text{[Proposition 3.2.23}(i)\text{]} \\
& = & \llparenthesis D \rrparenthesis [\llparenthesis \vec{s} \rrparenthesis]. & \text{[Theorem 3.2.25}(i)\text{]}
\end{array}$$

ii) Both substitutions, $[\![\vec{v} := \vec{t}]\!]$ and $[\![\llcorner \vec{v} \lrcorner := \llcorner \vec{t} \lrcorner]\!]$ are legal substitutions, the former in $\mathcal{H}_{\lambda c}$ and the latter in λc. The proof uses the fact that lifting and projection are each others inverse:

$$\begin{array}{rcll} \llcorner s\,[\![\vec{v} := \vec{t}]\!]\,\lrcorner & = & \llcorner \ulcorner \llcorner s \lrcorner \urcorner\,[\![\ulcorner \llcorner \vec{v} \lrcorner \urcorner := \ulcorner \llcorner \vec{t} \lrcorner \urcorner]\!] \lrcorner & \text{[Theorem 3.2.25}(ii)\text{]} \\ & = & \llcorner \ulcorner \llcorner s \lrcorner\,[\![\llcorner \vec{v} \lrcorner := \llcorner \vec{t} \lrcorner]\!] \urcorner \lrcorner & \text{[Proposition 3.2.23}(ii)\text{]} \\ & = & \llcorner s \lrcorner\,[\![\llcorner \vec{v} \lrcorner := \llcorner \vec{t} \lrcorner]\!]\,. & \text{[Theorem 3.2.25}(i)\text{]} \end{array}$$

QED

The contractions of λc and $\mathcal{H}_{\lambda c}$ coincide. When comparing the contractions and rewrite steps of λc and $\mathcal{H}_{\lambda c}$, it is important to keep in mind that the rewrite relations of λc and $\mathcal{H}_{\lambda c}$ are defined differently. For the general idea, see Intermezzo 3.2.10.

When translating contractions $L \to_\imath R$ and $(l\vec{s})\downarrow_\beta \to_{\mathsf{i}} (r\vec{s})\downarrow_\beta$ to each other, the form of l, r and $\vec{s}$ plays an important role. The arguments s_i will each be of the form $s_i \equiv \vec{u}.\,t_i$ with $\vec{u}, t_i \in \textsc{Ter}(\mathcal{H}_{\lambda c})$ (see Remark 3.2.21). Because β-reduction respects types and $\overline{\eta}$-normal forms, $(l\vec{s})\downarrow_\beta$ and $(r\vec{s})\downarrow_\beta$ will be elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$ and their projections will be λc-terms.

Remark 3.2.27 The coincidence of the contractions (and later, the rewrite steps) in λc and $\mathcal{H}_{\lambda c}$ means not only that a lifted (or, projected) contraction is again a contraction, but also that it is a contraction of the same instance of the corresponding rewrite rule. For example, if

$$U \equiv (\delta h.\,h\,\langle x\rangle)\,\lceil \Lambda x.\,x\rceil \to_{\mathit{fill}} (\Lambda x.\,x)\,\langle x\rangle \equiv V$$

with 1 as the (omitted) index of $\lceil\ \rceil_1$ and δ_1 in the applied instance of the rewrite rule (*fill*), then

$$\ulcorner U \urcorner \;=\; \mathsf{hf}\,(\mathsf{habs}(h.\,\mathsf{mapp}\;h\;x))\;\mathsf{mabs}(x.\,x) \to_{\mathsf{fill}} \mathsf{mapp}\,(\mathsf{mabs}(x.\,x))\;x = \ulcorner V \urcorner$$

with the same (omitted) index 1 in hf_1 and habs_1 in the instance of the corresponding rewrite rule (fill). In the proofs we will not be explicit about this, but the reader should keep this in mind.

Notation. Let $\imath$ be (an instance of) a rewrite rule of λc. (The instance of) the rewrite rule of $\mathcal{H}_{\lambda c}$ corresponding to $\imath$ will be denoted by $\ulcorner \imath \urcorner$. Analogously, the rewrite rule of λc corresponding to the rewrite rule (i) of $\mathcal{H}_{\lambda c}$ will be denoted by $\llcorner \mathsf{i} \lrcorner$. Moreover, we assume the correspondence between the rewrite rules of λc and $\mathcal{H}_{\lambda c}$ is obvious.

Proposition 3.2.28 (Lifting contractions) *If $L \to_\imath R$ is a contraction in λc, then $\ulcorner L \urcorner \to_{\mathsf{i}} \ulcorner R \urcorner$ is a contraction of the corresponding rewrite rule in $\mathcal{H}_{\lambda c}$.*

Proof: See Figure 3.4. The proof is conducted by case analysis on a rewrite rule ($\imath$) of λc. We treat only two cases:

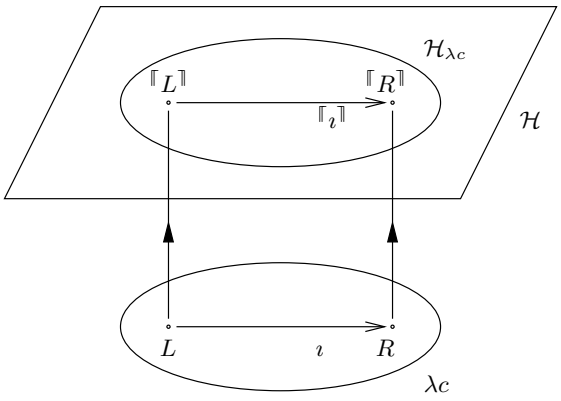


Figure 3.4: Lifting contractions

(ⓜβ)**:** Then,

$$\begin{array}{lll} \llbracket L \rrbracket & \equiv & \llbracket\ (\Lambda \vec{u}.\, U)\, \langle \vec{V} \rangle\ \rrbracket \\ & = & \mathsf{mapp}\ (\mathsf{mabs}(\vec{u}.\, \llbracket U \rrbracket))\ \llbracket \vec{V} \rrbracket \qquad \text{[def. lifting]} \\ & \twoheadleftarrow_\beta & (z, \vec{z}.\, \mathsf{mapp}\ (\mathsf{mabs}(\vec{v}.\, z\vec{v}))\ \vec{z}) \\ & & (\vec{u}.\, \llbracket U \rrbracket)\ \llbracket \vec{V} \rrbracket\ . \end{array}$$

By replacing the left-hand side of the rewrite rule (mb) by the right-hand side, we obtain

$$\begin{array}{lll} (z, \vec{z}.\, z\vec{z})(\vec{u}.\, \llbracket U \rrbracket)\llbracket \vec{V} \rrbracket & & \\ & \twoheadrightarrow_\beta & \llbracket U \rrbracket \llbracket \vec{u} := \llbracket \vec{V} \rrbracket \rrbracket \quad \text{[Lemma 3.2.14]} \\ & = & \llbracket U\, \llbracket \vec{u} := \vec{V} \rrbracket\ \rrbracket \quad \text{[Prop. 3.2.23}(ii)\text{]} \\ & \equiv & \llbracket R \rrbracket. \end{array}$$

(∘)**:** For the sake of readability, we consider only an instance of the collection of the composition rewrite rules where $n = 1$ in $\circ_n$ (and where consequently the first argument is an abstraction over only one variable).

Then,

$$\begin{array}{lll} \llbracket L \rrbracket & & \\ \equiv & \llbracket\ (\delta u.\, U) \circ (\Lambda \vec{v}.\, \delta \vec{v}\,'.\, V)\ \rrbracket & \\ = & \mathsf{comp}\ (\mathsf{habs}(u.\, \llbracket U \rrbracket))\ (\mathsf{mabs}(\vec{v}.\, \mathsf{habs}(\vec{v}\,'.\, \llbracket V \rrbracket))) & \text{[def. lifting]} \\ \twoheadleftarrow_\beta & (z, z'.\, \mathsf{comp}\ (\mathsf{habs}(u'.\, zu')) & \\ & \qquad (\mathsf{mabs}(\vec{w}.\, \mathsf{habs}(\vec{w}\,'.\, z'\vec{w}\vec{w}\,')))) & \\ & (u.\, \llbracket U \rrbracket)(\vec{v}, \vec{v}\,'.\, \llbracket V \rrbracket). & \end{array}$$

By replacing the left-hand side of the rewrite rule (mb) by the right-hand side, we obtain

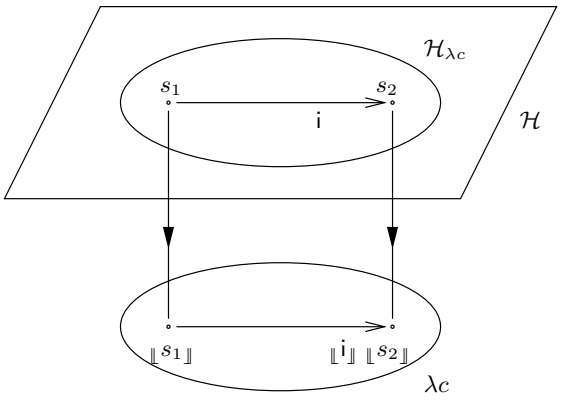


Figure 3.5: Projecting contractions

$$
\begin{array}{lll}
 & (z, z'.\, \mathsf{habs}(\vec{w}\,'.\, z(\mathsf{mabs}(\vec{w}.\, z'\vec{w}\vec{w}\,')))) & \\
 & (u.\, \ulcorner U \urcorner)(\vec{v}, \vec{v}\,'.\, \ulcorner V \urcorner) & \\
\twoheadrightarrow_\beta & \mathsf{habs}(\vec{v}\,'.\, \ulcorner U \urcorner\, [\![u := \mathsf{mabs}(\vec{v}.\, \ulcorner V \urcorner)]\!]) & \text{[Lemma 3.2.14]} \\
= & \mathsf{habs}(\vec{v}\,'.\, \ulcorner U \urcorner\, [\![u := \ulcorner\, \Lambda\vec{v}.\, V\, \urcorner]\!]) & \text{[def. lifting]} \\
= & \mathsf{habs}(\vec{v}\,'.\, \ulcorner U\, [\![u := \Lambda\vec{v}.\, V]\!]\, \urcorner) & \text{[Prop. 3.2.23}(ii)\text{]} \\
= & \ulcorner\, \delta\vec{v}\,'.\, U[\![u := \Lambda\vec{v}.\, V]\!]\, \urcorner & \text{[def. lifting]} \\
\equiv & \ulcorner R \urcorner. &
\end{array}
$$

QED

Proposition 3.2.29 (Projecting contractions) *If $s_1 \to_{\mathsf{i}} s_2$ is a contraction in $\mathcal{H}_{\lambda c}$, then $\llbracket s_1 \rrbracket \to_\imath \llbracket s_2 \rrbracket$ is a contraction of the corresponding rewrite rule in λc.*

Proof: See Figure 3.5. The proof is conducted by case analysis on a rewrite rule (i) of $\mathcal{H}_{\lambda c}$. We treat only two cases:

(mb)**:** Because it is an element of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, s_1 is of the form $\mathsf{mapp}\,(\mathsf{mabs}(\vec{u}.\, t))\ \vec{t}$ with $t, \vec{t} \in \textsc{Ter}(\mathcal{H}_{\lambda c})$. In that case, s_2 is of the form $t[\![\vec{u} := \vec{t}]\!]$.

Since t and $\vec{t}$ are also elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, they can be projected to λc-terms. Then the whole contraction can be directly projected to a contraction of the rewrite rule $(\textcircled{\scriptsize m}\beta)$ in λc:

$$
\begin{array}{lll}
\llbracket\, \mathsf{mapp}\,(\mathsf{mabs}(\vec{u}.\, t))\ \vec{t}\, \rrbracket & = & (\Lambda\vec{u}.\, \llbracket t \rrbracket)\langle \llbracket \vec{t} \rrbracket \rangle \quad \text{[def. projection]} \\
 & \to_{\textcircled{\scriptsize m}\beta} & \llbracket t \rrbracket [\![\vec{u} := \llbracket \vec{t} \rrbracket]\!] \\
 & = & \llbracket t\, [\![\vec{u} := \vec{t}]\!]\, \rrbracket. \quad \text{[Prop. 3.2.26}(ii)\text{]}
\end{array}
$$

(cmp)**:** Like in the proof of the previous proposition, only an instance of the rewrite rule (comp) with arity 2 is considered.

Because it is an element of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, s_1 is of the form $\mathsf{comp}\ (\mathsf{habs}(u.t))\ (\mathsf{mabs}(\vec{v}.\mathsf{habs}(\vec{v}'.t')))$ with $t, t' \in \textsc{Ter}(\mathcal{H}_{\lambda c})$. In that case, s_2 is of the form $\mathsf{habs}(\vec{v}'.t[\![u := \mathsf{mabs}(\vec{v}.t')]\!])$.

Since t and t' are also elements of $\textsc{Ter}(\mathcal{H}_{\lambda c})$, they can be projected to λc-terms. Then the whole contraction can be directly projected to a contraction of the rewrite rule $(\circ)$ in λc:

$$
\begin{array}{rll}
& \llbracket\ \mathsf{comp}\ (\mathsf{habs}(u.t))\ (\mathsf{mabs}(\vec{v}.\mathsf{habs}(\vec{v}'.t')))\ \rrbracket & \\
= & (\delta u.\ \llbracket t \rrbracket) \circ (\Lambda \vec{v}.\delta \vec{v}'.\ \llbracket t' \rrbracket) & \text{[definition of projection]} \\
\to_{\circ} & \delta \vec{v}'.\ \llbracket t \rrbracket\ [\![u := \Lambda \vec{v}.\ \llbracket t' \rrbracket]\!] & \\
= & \delta \vec{v}'.\ \llbracket t \rrbracket\ [\![u := \llbracket\ \mathsf{mabs}(\vec{v}.t')\ \rrbracket]\!] & \text{[definition of projection]} \\
= & \delta \vec{v}'.\ \llbracket t\ [\![u := \mathsf{mabs}(\vec{v}.t')]\!]\ \rrbracket & \text{[Proposition 3.2.26}(ii)\text{]} \\
= & \llbracket\ \mathsf{habs}(\vec{v}'.t[\![u := \mathsf{mabs}(\vec{v}.t')]\!])\ \rrbracket. & \text{[definition of projection]}
\end{array}
$$

QED

The one–to–one correspondence culminates in the next two theorems where it is shown that rewrite steps and sequences can be lifted or projected between the context calculus and the subsystem $\mathcal{H}_{\lambda c}$.

Theorem 3.2.30 *Let $U \in \textsc{Ter}(\lambda c)$. If $U \to_{\imath_1} U_1 \to_{\imath_2} \ldots \to_{\imath_n} U_n$ then $\llparenthesis U \rrparenthesis \to_{\llparenthesis \imath_1 \rrparenthesis} \llparenthesis U_1 \rrparenthesis \to_{\llparenthesis \imath_2 \rrparenthesis} \ldots \to_{\llparenthesis \imath_n \rrparenthesis} \llparenthesis U_n \rrparenthesis$.*

Proof: Only the case $n = 1$ is treated; the statement follows by induction on the length of the rewrite sequence.
Let $U \equiv C[L] \to C[R] \equiv V$, where $L \to_\imath R$ is an $\imath$-contraction in λc. Then

$$
\begin{array}{rll}
\llparenthesis U \rrparenthesis & \equiv & \llparenthesis C[L] \rrparenthesis \\
& = & \llparenthesis C \rrparenthesis [\llparenthesis L \rrparenthesis] \quad \text{[Proposition 3.2.23}(i)\text{]} \\
& \to_{\llparenthesis \imath \rrparenthesis} & \llparenthesis C \rrparenthesis [\llparenthesis R \rrparenthesis] \quad \text{[Proposition 3.2.28]} \\
& = & \llparenthesis C[R] \rrparenthesis \quad \text{[Proposition 3.2.23}(i)\text{]} \\
& \equiv & \llparenthesis V \rrparenthesis.
\end{array}
$$

QED

Theorem 3.2.31 *Let $s \in \textsc{Ter}(\mathcal{H}_{\lambda c})$. If $s \to_{\mathsf{i}_1} s_1 \to_{\mathsf{i}_2} \ldots \to_{\mathsf{i}_n} s_n$ then $\llbracket s \rrbracket \to_{\llbracket \mathsf{i}_1 \rrbracket} \llbracket s_1 \rrbracket \to_{\llbracket \mathsf{i}_2 \rrbracket} \ldots \to_{\llbracket \mathsf{i}_n \rrbracket} \llbracket s_n \rrbracket$.*

Proof: Only the case $n = 1$ is treated; the statement follows by induction on the length of the rewrite sequence. The case $n = 1$ is proven analogously to the proof of the previous proposition, using complementary propositions.
Let $s \equiv D[s'] \to_{\mathsf{i}} D[t'] \equiv t$, where $s' \to_{\mathsf{i}} t'$ is an (i)-contraction in $\mathcal{H}_{\lambda c}$. Then

$$
\begin{array}{rll}
\llbracket s \rrbracket & \equiv & \llbracket D[s'] \rrbracket \\
& = & \llbracket D \rrbracket [\llbracket s' \rrbracket] \quad \text{[Proposition 3.2.26}(i)\text{]} \\
& \to_{\llbracket \mathsf{i} \rrbracket} & \llbracket D \rrbracket [\llbracket t' \rrbracket] \quad \text{[Proposition 3.2.29]} \\
& = & \llbracket D[t'] \rrbracket \quad \text{[Proposition 3.2.26}(i)\text{]} \\
& \equiv & \llbracket t \rrbracket.
\end{array}
$$

QED

In conclusion, the context calculus λc is $\mathcal{H}_{\lambda c}$.

3.2.4 Commutation of rewriting in λc

Recall that the commutation property of the pairs of rewrite relations states that, if $b \mathrel{{}_\imath\!\!\twoheadleftarrow} a \mathrel{\twoheadrightarrow_\jmath} c$ then there is d such that $b \mathrel{\twoheadrightarrow_\jmath} d \mathrel{{}_\imath\!\!\twoheadleftarrow} c$. In the context calculus λc, the commutation property of pair of rewrite relations is a consequence of the orthogonality property of $\mathcal{H}$ and the two correspondences: between $\mathcal{H}$ and $\mathcal{H}_{\lambda c}$ on the one hand, and on the other hand, between $\mathcal{H}_{\lambda c}$ and λc. We will first show that in $\mathcal{H}$ each pair of rewrite relations commutes, then that $\mathcal{H}_{\lambda c}$ has the same property and consequently, the context calculus λc too.

The proof that in $\mathcal{H}$ each pair of rewrite relations commutes relies on tracing the descendants of redexes along a rewrite sequence and reducing them simultaneously. More concretely, the proof focuses on tracing the descendants of redexes involved in a pair of diverging rewrite sequences and reducing these descendants simultaneously in the pair of converging rewrite sequences.

Tracing descendants of a set of redexes along a rewrite sequence is not a trivial task. First, the definition of descendant relation is complex (cf. Definition 1.3.26). The definition of rewrite steps includes β-reduction, next to the replacement of the left-hand side of a rewrite rule by the right-hand side. Consequently, the descendant relation over a rewrite step in higher-order rewrite systems includes the descendant relation over β-reduction. Second, the difficulty with descendants of a set of redexes in a higher-order term rewrite system is that the descendants of mutually disjoint redexes (i.e. redexes situated in different subterms of a term) can become nested (i.e. one descendant becomes a subterm of an another one) after a rewrite step, as for example in

$$(\lambda x.\, \underline{(\lambda y.\, x)\, y'})\, (\overline{(\lambda z.\, z)\, z'}) \to_\beta \underline{(\lambda y.\, (\overline{(\lambda z.\, z)\, z'}))\, y'}$$

in lambda calculus (see also Example 1.3.27 where an example in PRSs is given). This is not the case in the first-order term rewrite systems.

Luckily, we work in *orthogonal* rewrite systems, where despite of these difficulties, strong results hold regarding redexes and their descendants. We list some of the well-known results, which we will need in the proof. The definitions of notions and results mentioned below were formulated in Section 1.3.

i) Descendants of a redex are again redexes. Moreover, they are redexes of the same rewrite rule as their antecedents.

ii) Redex occurrences in a term are independent, that is, contraction of one redex occurrence does not disturb contraction of another redex occurrence. Then, any set of redexes in a term can be extracted independently of each other and reduced simultaneously.

The notion that is concerned with simultaneously reducing a set of pairwise independent redexes is called complete development (see Definition 1.3.21). A complete development step is denoted by $s \mathrel{-\!\!\circ\!\!\to} t$.

Notation. If $\mathcal{V}$ and $\mathcal{U}$ are two sets of pairwise independent redexes, then the set $\mathcal{V}/\mathcal{U}$ denotes the set of descendants of $\mathcal{V}$ over the complete development of $\mathcal{U}$. Let

$-\!\circ\!\twoheadrightarrow$ denote the reflexive–transitive closure of $-\!\circ\!\rightarrow$. Let $s \; -\!\circ\!\rightarrow^{\mathcal{V}} \; t$ denote a complete development step of the set of pairwise independent redexes $\mathcal{V}$.

We formulate two definitions about rewriting in λc and the pattern rewrite systems $\mathcal{H}$ and $\mathcal{H}_{\lambda c}$.

Definition 3.2.32 (Uniform rewriting)

i) A rewrite sequence $s \twoheadrightarrow_\imath t$ will be called uniform.

ii) A complete development of a set of pairwise independent redexes of the rewrite rule ($\imath$) will be called a uniform complete development and it will be denoted by $-\!\circ\!\rightarrow_\imath$.

Definition 3.2.33 (Tiles)

i) A diagram $d \twoheadleftarrow_\jmath b \twoheadleftarrow_\imath a \twoheadrightarrow_\jmath c \twoheadrightarrow_\imath d$ will be called a commutation tile in both λc and pattern rewrite systems (see Figure 3.6(a)).

ii) A diagram $d \leftarrow\!\circ\!- b \leftarrow\!\circ\!- a -\!\circ\!\rightarrow c -\!\circ\!\rightarrow d$ will be called a cd-tile (see Figure 3.6(b)).

iii) A diagram $d \leftarrow\!\circ\!-_\jmath b \leftarrow\!\circ\!-_\imath a -\!\circ\!\rightarrow_\jmath c -\!\circ\!\rightarrow_\imath d$ will be called a commutation cd-tile (see Figure 3.6(c)).

The proof of the commutation property of rewriting in $\mathcal{H}$ rests on the Prism theorem (cf. [Oos95]). This theorem offers the basis for existence of commutation cd-tiles in $\mathcal{H}$.

Theorem 3.2.34 (Prism theorem)

i) *Let* $t \leftarrow\!\circ\!-^{\mathcal{U}} s -\!\circ\!\rightarrow^{\mathcal{V}} t'$ *be diverging complete developments of independent redexes* $\mathcal{U} \subseteq \mathcal{V}$. *Then* $t \; -\!\circ\!\rightarrow^{\mathcal{V}/\mathcal{U}} \; t'$ *and the descendant relations induced by* $-\!\circ\!\rightarrow^{\mathcal{U}}; -\!\circ\!\rightarrow^{\mathcal{V}/\mathcal{U}}$ *and* $-\!\circ\!\rightarrow^{\mathcal{V}}$ *are the same.*

ii) *If* $\mathcal{U} \cup \mathcal{V}$ *is independent, then* $\mathcal{V}/\mathcal{U}$ *is independent.*

iii) *Every development step can be serialised, i.e. if* $s -\!\circ\!\rightarrow t$ *then there exists a rewrite sequence* $s \twoheadrightarrow t$ *consisting of rewrite steps, inducing the same descendant relation.*

The next four results deal with tiles in $\mathcal{H}$, $\mathcal{H}_{\lambda c}$ and finally in λc. The first proposition claims that in $\mathcal{H}$ any pair of diverging uniform complete development steps can be finished into a commutation cd-tile (see Figure 3.6(c)). The proof uses the orthogonality property of $\mathcal{H}$ and the Prism theorem.

Proposition 3.2.35 (Commutation cd-tiles in $\mathcal{H}$) *Let* s, t, t' *be terms of* $\mathcal{H}$. *Suppose* $s \; -\!\circ\!\rightarrow_{\mathsf{i}} \; t$ *and* $s \; -\!\circ\!\rightarrow_{\mathsf{j}} \; t'$. *Then there is a term* t'' *of* $\mathcal{H}$ *such that* $t' \; -\!\circ\!\rightarrow_{\mathsf{i}} \; t''$ *and* $t \; -\!\circ\!\rightarrow_{\mathsf{j}} \; t''$.

a) commutation tile *b*) cd-tile *c*) commutation cd-tile

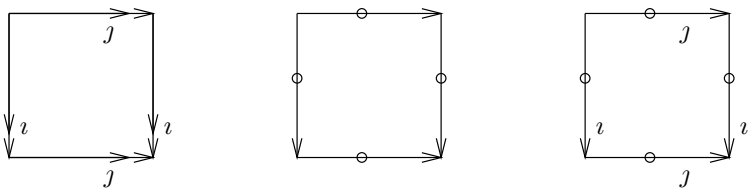


Figure 3.6: Tiles

Proof: If $s \multimap\!\!\to_{\mathsf{i}} t$ and $s \multimap\!\!\to_{\mathsf{j}} t'$ then these complete developments are complete developments of certain sets of independent redexes, say $\mathcal{U}$ and $\mathcal{U}'$, each set containing redexes of the rewrite rule (i) and (j), respectively. Thus, $s \multimap\!\!\to^{\mathcal{U}} t$ and $s \multimap\!\!\to^{\mathcal{U}'} t'$. Let $\mathcal{V} = \mathcal{U} \cup \mathcal{U}'$. By the Prism theorem 3.2.34(i), there is a t'' of $\mathcal{H}$ such that $s \multimap\!\!\to^{\mathcal{V}} t''$, $t \multimap\!\!\to^{\mathcal{V}/\mathcal{U}} t''$ and $t' \multimap\!\!\to^{\mathcal{V}/\mathcal{U}'} t''$. Since the redexes of $\mathcal{U}$ have been reduced in the rewrite sequence $s \multimap\!\!\to^{\mathcal{U}} t$, the reduction $t \multimap\!\!\to^{\mathcal{V}/\mathcal{U}} t''$ consists of a complete development of residuals of $\mathcal{V} \setminus \mathcal{U} = \mathcal{U}' \setminus \mathcal{U}$. Since residuals of independent redexes of the rewrite rule (j) are again independent redexes of the same rewrite rule (by Prism theorem 3.2.34(ii)), the reduction $t \multimap\!\!\to^{\mathcal{V}/\mathcal{U}} t''$ is a complete development of independent redexes of the rewrite rule (j). Thus, $t \multimap\!\!\to_{\mathsf{j}} t''$.

Analogously for $t' \multimap\!\!\to_{\mathsf{i}} t''$. QED

The second proposition states that in $\mathcal{H}$ any pair of diverging uniform rewrite sequences can be finished into a commutation tile. The proof uses the Prism theorem and the fact that each rewrite step is a complete development of a singleton.

Proposition 3.2.36 (Commutation tiles in $\mathcal{H}$) *Let $s, t, t' \in \textsc{Ter}(\mathcal{H})$, and let* (i) *and* (j) *be rewrite rules of $\mathcal{H}$. If $t \mathrel{\text{⇚}}_{\mathsf{i}} s \twoheadrightarrow_{\mathsf{j}} t'$, then there is $s' \in \textsc{Ter}(\mathcal{H})$ such that $t \twoheadrightarrow_{\mathsf{j}} s' \mathrel{\text{⇚}}_{\mathsf{i}} t'$.*

Proof: The proof is depicted in Figure 3.7. We first prove that if $s \to_{\mathsf{i}} t_1$ and $s \twoheadrightarrow_{\mathsf{j}} t'$, then there is a term s'_1 of $\mathcal{H}$ such that $t' \twoheadrightarrow_{\mathsf{i}} s'_1$ and $t_1 \twoheadrightarrow_{\mathsf{i}} s'_1$. Since each rewrite step is a complete development of a singleton, these diverging rewrite sequences can be translated to uniform complete developments $s \multimap\!\!\to_{\mathsf{i}} t_1$ and $s \multimap\!\!\to\!\!\!\to_{\mathsf{j}} t'$. Now the diagram can be finished by tiling with commutation cd-tiles (proved by induction to the length of $s \multimap\!\!\to\!\!\!\to_{\mathsf{j}} t'$ and using Proposition 3.2.35; in the figure the application of the induction step is denoted by $(!)_1$). In this way we get a term s'_1 of $\mathcal{H}$ such that $t' \multimap\!\!\to_{\mathsf{i}} s'_1$ and $t_1 \multimap\!\!\to\!\!\!\to_{\mathsf{j}} s'_1$. Since complete developments can be serialised (Theorem 3.2.34), we have $t' \twoheadrightarrow_{\mathsf{i}} s'_1$ and $t_1 \twoheadrightarrow_{\mathsf{j}} s'_1$.

Then, by induction to the length of $s \twoheadrightarrow_{\mathsf{i}} t$, there is $s' \in \textsc{Ter}(\mathcal{H})$ such that $t \twoheadrightarrow_{\mathsf{j}} s' \mathrel{\text{⇚}}_{\mathsf{i}} t'$ (in the figure, the application of the induction step is denoted by $(!)_2$). QED

The third proposition says that now in $\mathcal{H}_{\lambda c}$ any pair of diverging uniform rewrite sequences can be finished into a commutation tile. The proof relies on the fact that $\mathcal{H}_{\lambda c}$ is a subsystem of $\mathcal{H}$, in which the same property holds.

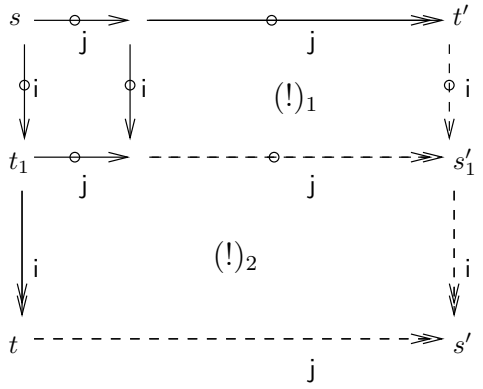


Figure 3.7: Proof of commutation of rewriting in $\mathcal{H}$

Proposition 3.2.37 (Commutation tiles in $\mathcal{H}_{\lambda c}$) *Let $s, t, t' \in \text{TER}(\mathcal{H}_{\lambda c})$ and $(\mathsf{i}), (\mathsf{j})$ be rewrite rules of $\mathcal{H}_{\lambda c}$. If $t \twoheadleftarrow_{\mathsf{i}} s \twoheadrightarrow_{\mathsf{j}} t'$, then there is $s' \in \text{TER}(\mathcal{H}_{\lambda c})$ such that $t \twoheadrightarrow_{\mathsf{j}} s' \twoheadleftarrow_{\mathsf{i}} t'$.*

Proof: Suppose $t \twoheadleftarrow_{\mathsf{i}} s \twoheadrightarrow_{\mathsf{j}} t'$. Because rewrite sequences of $\mathcal{H}_{\lambda c}$ are also rewrite sequences of $\mathcal{H}$, there is a s' in $\mathcal{H}$ such that $t' \twoheadrightarrow_{\mathsf{i}} s'$ and $t \twoheadrightarrow_{\mathsf{j}} s'$, by Proposition 3.2.36. Because $\mathcal{H}_{\lambda c}$ is closed under rewriting, this diagram is entirely within $\mathcal{H}_{\lambda c}$. QED

Finally, the fourth result states that commutation tiles exist in λc, that is, any pair of rewrite relations in λc commutes.

Theorem 3.2.38 (Commutation tiles in λc) *Let $U, V, V' \in \text{TER}(\lambda c)$ and let $\imath$ and $\jmath$ be two rewrite rules of λc. If $V \twoheadleftarrow_{\imath} U \twoheadrightarrow_{\jmath} V'$, then there is $W \in \text{TER}(\lambda c)$ such that $V \twoheadrightarrow_{\jmath} W \twoheadleftarrow_{\imath} V'$.*

Proof: The proof is depicted in Figure 3.8. Each pair of diverging reductions $U \twoheadrightarrow_{\imath} V_1$ and $U \twoheadrightarrow_{\jmath} V_2$ of λc can be lifted to a pair of diverging reductions $\llbracket U \rrbracket \twoheadrightarrow_{\llbracket \imath \rrbracket} \llbracket V_1 \rrbracket$ and $\llbracket U \rrbracket \twoheadrightarrow_{\llbracket \jmath \rrbracket} \llbracket V_2 \rrbracket$, respectively, in $\mathcal{H}_{\lambda c}$, by Theorem 3.2.30. By Proposition 3.2.37 there is a pair of converging reductions $\llbracket V_1 \rrbracket \twoheadrightarrow_{\llbracket \jmath \rrbracket} t$ and $\llbracket V_2 \rrbracket \twoheadrightarrow_{\llbracket \imath \rrbracket} t$ in $\mathcal{H}_{\lambda c}$. The converging reductions can be projected back to converging reductions from V_1 and V_2, i.e. $V_1 = \llparenthesis \llbracket V_1 \rrbracket \rrparenthesis \twoheadrightarrow_{\jmath} \llparenthesis t \rrparenthesis$ and $V_2 = \llparenthesis \llbracket V_2 \rrbracket \rrparenthesis \twoheadrightarrow_{\imath} \llparenthesis t \rrparenthesis$ respectively, in λc, by Theorem 3.2.31 and Theorem 3.2.25. QED

We conclude this section with a result over unions of rewrite relations in λc.

Proposition 3.2.39 *Let $U, V, V' \in \text{TER}(\lambda c)$. Let $\to_1$ and $\to_2$ denote two arbitrary unions of the rewrite relations of λc. If $V \twoheadleftarrow_1 U \twoheadrightarrow_2 V'$, then there is $W \in \text{TER}(\lambda c)$ such that $V \twoheadrightarrow_2 W \twoheadleftarrow_1 V'$.*

Proof: By Theorem 3.2.38 each pair of rewrite relations in λc commute with each other. Then the statement follows by a corollary of the Hindley–Rosen lemma, Corollary 1.1.12. QED

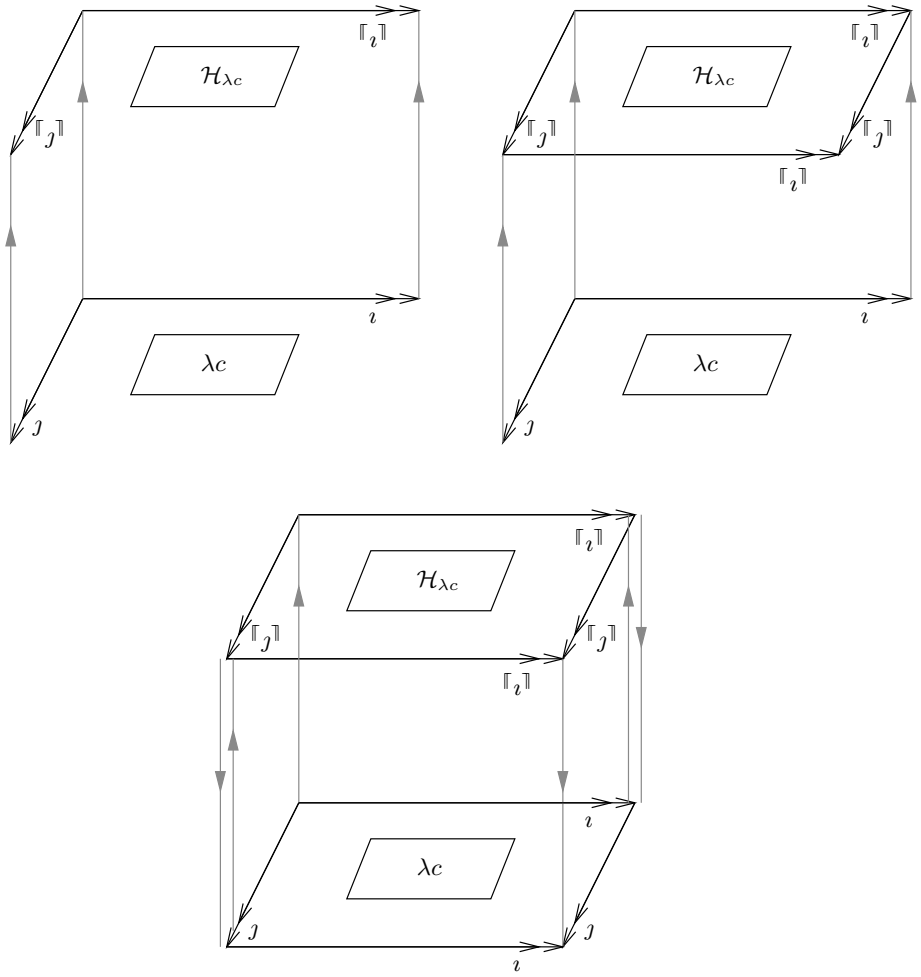


Figure 3.8: Proof of commutation of rewriting in λc

In particular, λ-calculus rewriting commutes with context-related rewriting. This result justifies our view of the context calculus λc as a two-level calculus, with issues regarding the lambda calculus separated from the issues regarding the extension.

3.2.5 Confluence of rewriting in λc

The confluence property of λc follows now easily. Recall that, the confluence property states that any pair of diverging rewrite sequences $b \twoheadleftarrow a \twoheadrightarrow c$ can be prolonged by a pair of converging rewrite sequences $b \twoheadrightarrow d \twoheadleftarrow c$. In this section we present a proof via the commutation property of each pair of the rewrite relations in λc and outline an alternative proof via the confluence property of $\mathcal{H}$.

Theorem 3.2.40 (Confluence of λc) *λc is confluent.*

Proof: The theorem follows directly from the commutation property of each pair of the rewrite relations in λc. The rewrite steps of the same rewrite rule in a pair of diverging rewrite sequences can be grouped. Such a diverging pair of rewrite sequences can be tiled to a diagram by commuting tiles, which exists due to the commutation property of pairs of rewrite relations in λc. QED

An alternative proof can be given via the confluence property of $\mathcal{H}$. In higher-order rewriting, an orthogonal pattern rewrite system is confluent. Since $\mathcal{H}$ is orthogonal, it is confluent. Since $\mathcal{H}_{\lambda c}$ is a subsystem of $\mathcal{H}$, it is confluent too. Finally, by the correspondence between $\mathcal{H}_{\lambda c}$ and λc, λc is also confluent. In fact, Figure 3.8 can be reused for this proof, by erasing the indices of the rewrite sequences.

3.3 Normalisation in the context calculus λc

At this point of theory development, three normalisation properties are of interest: the weak and strong normalisation (Definition 1.1.6), and the preservation of strong normalisation (Definition 1.2.28). A rewrite system has the weak normalisation property if from each element there is a finite rewrite sequence to a normal form, that is, to a term that cannot be rewritten any further. A rewrite system has the strong normalisation property if there are no infinite rewrite sequences. Hence, the strong normalisation implies the weak normalisation. A rewrite system that is an extension of lambda calculus has the property of preservation of strong normalisation if every λ-term that is strongly normalising with respect to β-rewriting is strongly normalising with respect to the rewriting of this extension. We will show here that the context calculus is not weakly normalising; hence also not strongly normalising. We will also show that, in a trivial way, the context calculus has the property of the preservation of strong normalisation.

The context calculus λc, in its untyped version, does not have the weak normalisation property. The absence of normalisation is caused by two sorts of terms. We discuss these terms in turn.

The first kind of non-normalising terms comes from the untyped lambda calculus. Recall that the context calculus is an extension of the untyped lambda calculus and the untyped lambda calculus is not weakly normalising. Then, because the context calculus is an extension of the untyped lambda calculus, the terms and rewrite steps of the untyped lambda calculus can be represented within λc. Because some untyped λ-terms do not have a (β-)normal form, their representations will not have a a (β-)normal form in λc either. An example of such a term is the well-known λ-term Ω, which has an infinite reduction:

$$\Omega \equiv (\lambda x.\, x x)(\lambda x.\, x x) \to_\beta \Omega.$$

Because this is the only rewrite step that can be performed starting from Ω, this term has no normal form, that is, it is not weakly normalising.

The second kind of non-normalising terms comes from the extension. That is, the context machinery introduces terms that are not weakly normalising with respect to the context-related rewriting. Examples of such terms can be found by studying the example above. For example, consider the terms Ω_{fill}, $\Omega_{\textcircled{\scriptsize m}\beta}$ and $\Omega'_{\textcircled{\scriptsize m}\beta}$ with their infinite reductions:

$$\Omega_{\mathit{fill}} \equiv (\delta h.\, h\lceil h\rceil)\lceil \delta h.\, h\lceil h\rceil\rceil \to_{\mathit{fill}} \Omega_{\mathit{fill}}$$

$$\Omega_{\textcircled{\scriptsize m}\beta} \equiv (\Lambda u.\, u\langle u\rangle)\langle \Lambda u.\, u\langle u\rangle\rangle \to_{\textcircled{\scriptsize m}\beta} \Omega_{\textcircled{\scriptsize m}\beta}$$

$$\Omega'_{\textcircled{\scriptsize m}\beta} \equiv (\Lambda u, v.\, u\langle u, v\rangle)\langle \Lambda u, v.\, u\langle u, v\rangle,\ \Lambda u, v.\, u\langle u, v\rangle\rangle \to_{\textcircled{\scriptsize m}\beta} \Omega'_{\textcircled{\scriptsize m}\beta}.$$

These terms and the term Ω are counterexamples of weak normalisation for a single rule.

The non-normalising terms of the first kind can only be ruled out by introducing types on terms (as well in the untyped lambda calculus as in its representation in λc). However, the non-normalising terms of the second kind will in particular be considered a nuisance, because the meta-operations on contexts in lambda calculus do have a result, and therefore the context-related rewriting in λc should at least be normalising.

However, the absence of weak or strong normalisation in λc at this point of theory development does not mean a drawback. The reason is that the context calculus in its untyped version contains superfluous elements, that is, elements which do not represent a meaningful object of lambda calculus. The terms Ω_{fill}, $\Omega_{\textcircled{\scriptsize m}\beta}$ and $\Omega'_{\textcircled{\scriptsize m}\beta}$ are examples of such elements. Such elements should be filtered out by syntactic typing. It is then up to a particular syntactic type system to eliminate the non-normalising terms, as well, and in particular the non-normalising terms of the second kind.

The property of preservation of strong normalisation is trivial to check in λc.

Theorem 3.3.1 (Preservation of strong normalisation) *Rewriting in λc has the property of preservation of strong normalisation.*

Proof: Let M be an untyped λ-term. This term can directly be represented within λc, namely as the term M itself. Because by rewriting of λc no other redexes than β-redexes can be introduced, this term is strongly normalising if it is strongly normalising in the untyped lambda calculus. QED

In the chapters that follow, we will return to the normalisation properties.

3.4 Context calculus for arbitrary term rewrite systems

The technique that has been employed in λc for formalisation of meta-contexts of lambda calculus can be applied to formalise meta-contexts of an arbitrary term rewrite system with[1] binders. That is, this technique can be parametrised over a term rewrite system.

Here, we present the definition of a context calculus parametrised over an arbitrary term rewrite system, and briefly comment on its properties. The definition is formulated in the format of higher-order rewrite systems, which are a uniform framework for term rewrite systems with binders. In particular we use pattern rewrite systems (PRSs, see Section 1.3). This format and the higher-order rewriting theory offer us a firm grip on the form and the properties of the context-related part of the definition.

Recall that the context calculus λc is an extension of lambda calculus (cf. Definition 3.1.5). The additional part, which is concerned with contexts, includes the following constructors: multiple abstraction Λ_n, multiple application $\langle\,\rangle_n$, hole abstraction δ_n, hole filling $\lceil\,\rceil_n$, and composition $\circ_n$. Furthermore, the additional part includes the context rewrite rules ($\textcircled{m}\beta$), (*fill*) and ($\circ$).

The syntax and the set of rewrite rules of an arbitrary term rewrite system $\mathcal{A}$ can be extended to deal with contexts in the same way as the lambda calculus has been extended to form λc.

Definition 3.4.1 (Context calculus $\mathcal{H}_c$ for $\mathcal{A}$)
Let $\mathcal{A}$ be a term rewrite system with binders and let $\mathcal{H} = \langle \mathcal{C}, \mathcal{R} \rangle$ be its representation in the higher-order (PRS) format with $\mathcal{C}$ the signature and $\mathcal{R}$ the set of rewrite rules. Then, the context calculus for $\mathcal{A}$ is defined as the PRS $\mathcal{H}_c = \langle \mathcal{C}_c, \mathcal{R}_c \rangle$ where $\mathcal{C}_c$ and $\mathcal{R}_c$ are defined as follows.

i) The signature $\mathcal{C}_c$ of $\mathcal{H}_c$ consists of the elements of $\mathcal{C}$ and the following function

[1] For formalisation of meta-contexts of an arbitrary term rewrite system without binders, see Intermezzo 2.2.4.

symbols ($n \in \mathbb{N}$):

$$\begin{array}{lcll} \mathsf{mabs}_n & : & (\vec{0} \to 0) \to 0 & \text{with } |\vec{0}| = n \\ \mathsf{mapp}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1 \\ \mathsf{habs}_n & : & (\vec{0} \to 0) \to 0 & \text{with } |\vec{0}| = n \\ \mathsf{hf}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1 \\ \mathsf{comp}_n & : & \vec{0} \to 0 & \text{with } |\vec{0}| = n+1. \end{array}$$

The additional function symbols are assumed to be disjoint from the function symbols of $\mathcal{H}$.

ii) The set of rewrite rules $\mathcal{R}_c$ of $\mathcal{H}_c$ consists of the rewrite rules $\mathcal{R}$ and the following rewrite rules, called the context rewrite rules:

$$z, \vec{z}.\, \mathsf{mapp}_n\ (\mathsf{mabs}_n(\vec{u}.\, z\vec{u}))\ \vec{z} \ \to\ z, \vec{z}.\, z\vec{z} \qquad (\mathsf{mb})$$

$$z, \vec{z}.\, \mathsf{hf}_n\ (\mathsf{habs}_n(\vec{u}.\, z\vec{u}))\ \vec{z} \ \to\ z, \vec{z}.\, z\vec{z} \qquad (\mathsf{fill})$$

$$\begin{aligned} &z, \vec{z}.\, \mathsf{comp}_n\ (\mathsf{habs}_n(\vec{u}.\, z\vec{u}))\ (\mathsf{mabs}(\vec{v}_1.\, \mathsf{habs}(\vec{v}\,'_1.\, z_1\vec{v}_1\vec{v}\,'_1)))\ldots \\ &\qquad\qquad (\mathsf{mabs}(\vec{v}_n.\, \mathsf{habs}(\vec{v}\,'_n.\, z_n\vec{v}_n\vec{v}\,'_n))) \\ &\to\ z, \vec{z}.\, \mathsf{habs}(\vec{v}\,'_1, \ldots, \vec{v}\,'_n.\, z(\mathsf{mabs}(\vec{v}_1.\, z_1\vec{v}_1\vec{v}\,'_1))\ldots(\mathsf{mabs}(\vec{v}_n.\, z_n\vec{v}_n\vec{v}\,'_n))) \end{aligned} \qquad (\mathsf{cmp})$$

with $|\vec{z}| = n$.

Remark 3.4.2 The additional function symbols and the additional rewrite rules are 'translations' into the higher-order format of the context-related constructors and rewrite rules of λc, respectively. We have already seen such a translation in the definition of $\mathcal{H}$ (Definition 3.2.3) in the confluence proof of λc.

We look into some properties of rewriting in $\mathcal{H}_c$.

The context rewrite rules constitute an orthogonal system, that is, these rules are left-linear and there are no critical pairs with respect to these rules. Hence, the rewriting with respect to the context rewrite rules is confluent.

Furthermore, the patterns of the context rewrite rules involve only the added, context-related function symbols. With the context-related function symbols being disjoint from the function symbols $\mathcal{C}$ of the term rewrite system, the context rewrite rules do not overlap with the rewrite rules $\mathcal{R}$ of the term rewrite system. That is, there are no critical pairs between the context rewrite rules and the rewrite rules $\mathcal{R}$ of the term rewrite system.

The rewriting in $\mathcal{H}_c$ is not weakly normalising, because the rewriting with respect to the context rewrite rules is not weakly normalising. For example, the terms Ω_{fill} and $\Omega_{\textcircled{m}\beta}$ of Section 3.3, which are counterexamples for the weak normalisation of λc, can be translated into $\mathcal{H}_c$. Because the rewriting in $\mathcal{H}_c$ is not weakly normalising, it is also not strongly normalising. However, as in the case of the context calculus λc, by adding types (strong) normalisation of the context-related rewriting can be obtained (see Chapter 4). In that case, the normalisation properties of $\mathcal{H}_c$ will depend solely on the normalisation properties of the term rewrite system rules $\mathcal{R}$ and on how they combine with the context rewrite rules.

3.5 Related work

3.5.1 Double role of higher-order rewriting in λc

Higher-order rewriting plays a double role in the context calculus λc. We have already identified one of the two roles in the confluence proof of λc, where it has been shown that λc as a rewrite system with binders *is* a (second-order) pattern rewrite system.

Here, we describe the second role of higher-order rewriting in λc. This role concerns the representation of holes and meta-contexts within λc. We describe this role by studying four formulations of β-reduction: two in lambda calculus, one in the context calculus λc and one in pattern rewrite systems. We first show how β-reduction generated by the rewrite rule schema (β) of lambda calculus can be restated using contexts. This reformulation of β-reduction is stated via a 'rule', which will be called ($\beta_{[]}$). Next, we formulate β-reduction in the context calculus λc by translating the 'rule' ($\beta_{[]}$). The translation of this rule will be called (*beta*). Finally, we show that the formulation of β-reduction using the 'rule' (*beta*) employs similar techniques as the formulation of rewrite relations in pattern rewrite systems.

In these four formulations of β-reduction two issues will play a role: a rewrite rule (schema) and the way of using the rule to generate the rewrite relation β.

Two formulations of β-reduction in lambda calculus We recall the rewrite rule schema (β) of lambda calculus:

$$(\lambda x.\, M)\, N \to M\, [\![x := N]\!]\,. \qquad (\beta)$$

The rewrite relation β is generated as follows:

- We have $M_1 \to_\beta M_2$ if $M_1 \equiv (\lambda x.\, M)\, N$ and $M_2 \equiv M[\![x := N]\!]$ for certain λ-terms M and N. Here and in the schema, M and N denote arbitrary λ-terms. It is assumed that the free occurrences of the variable x in M are bound by the binder λx of the redex. It is because of such a treatment of M and N that they are called meta-variables and (β) is a schema: M and N in the schema are not substituted by λ-terms, but they are literally replaced by λ-terms.
- Furthermore, it is assumed that the steps are closed under meta-contexts: if $M_1 \to_\beta M_2$ then $C[M_1] \to_\beta C[M_2]$.

Note that $M[\![x := N]\!]$ denotes the *result* of applying the substitution $[\![x := N]\!]$ to M.

An example of a β-step (with the trivial meta-context) is

$$(\lambda x.\, x\, y)\, y \to_\beta (x\, y)[\![x := y]\!] = y\, y.$$

In fact, one could consider the left-hand side and the right-hand side of the rewrite rule schema (β) as meta-contexts with two holes

$$(\lambda x.\, []_1)\, []_2 \to []_1^{[\![x:=[]_2]\!]}. \qquad (\beta_{[]})$$

Here, the substitution $[\![x := []_2]\!]$ remains explicitly denoted as the label of the hole $[]_1$. Then the rewrite relation β can be generated as follows:

- We have $M_1 \to_\beta M_2$ if $M_1 \equiv (\lambda x.M)N \equiv ((\lambda x.[]_1)[]_2)[M,N]$ and $M_2 \equiv M[\![x := N]\!] \equiv []_1^{[\![x:=[]_2]\!]}[M,N]$ for certain λ-terms M and N.

  Note that the transformations $(\lambda x.M)N \equiv ((\lambda x.[]_1)[]_2)[M,N]$ and $M[\![x := N]\!] \equiv []_1^{[\![x:=[]_2]\!]}[M,N]$ involve extraction of the left-hand side and the right-hand side of $(\beta_{[]})$ from M_1 and M_2, respectively. The extraction is performed by the meta-operation of hole filling. Moreover, note that the transformation $\to_{\beta_{[]}}$ from $((\lambda x.[]_1)[]_2)[M,N]$ to $[]_1^{[\![x:=[]_2]\!]}[M,N]$ is the literal replacement of the left-hand side by the right-hand side of $(\beta_{[]})$.

- Furthermore, as in the case of the formulation via the schema (β), it is assumed that the steps are closed under meta-contexts: if $M_1 \to_\beta M_2$ then $C[M_1] \to_\beta C[M_2]$.

For example,

$$(\lambda x.xy)y \equiv ((\lambda x.[]_1)[]_2)[xy,y] \to_{\beta_{[]}} ([]_1[\![x := []_2]\!])[xy,y] \equiv (xy)[\![x := y]\!] = yy.$$

Formulation of β-reduction in the context calculus In the context calculus λc, terms and meta-context of lambda calculus can be represented. Hence, the 'rule' $(\beta_{[]})$ can be translated into a 'rule' involving λc-terms that represent the contexts of the left-hand side and the right-hand side of $(\beta_{[]})$:

$$\delta h_1, h_2.(\lambda x.h_1\langle x\rangle)h_2\langle\rangle \to \delta h_1, h_2.h_1\langle h_2\langle\rangle\rangle\,. \qquad (beta)$$

Here, the explicit substitution at the first hole $[]_1[\![x := []_2]\!]$ of the 'rule' $(\beta_{[]})$ is internalised by the communication mechanism of λc: viz. $h_1\langle h_2\langle\rangle\rangle = (h_1\langle x\rangle)[\![x := h_2\langle\rangle]\!]$.

The rewrite relation β can be generated on representations of λ-terms within λc by using $(beta)$ as follows:

- Let U_1 and U_2 be representations[2] of two λ-terms within λc. Then $U_1 \to_\beta U_2$ if $U_1 = (\delta h_1, h_2.(\lambda x.h_1\langle x\rangle)h_2\langle\rangle)\lceil V_1, V_2\rceil$ and $U_2 = (\delta h_1, h_2.h_1\langle h_2\langle\rangle\rangle)\lceil V_1, V_2\rceil$ for certain λc-terms V_1 and V_2. The λc-terms V_1 and V_2 are representations of certain communicating λ-terms, where the intended variable capturing is made explicit. The transformations, denoted here by $=$, involve extraction of the left-hand side and the right-hand side of $(beta)$ from U_1 and U_2, respectively. The extraction is performed by the (inverse) context-related rewrite relations. Moreover, note that the transformation $\to_{beta}$ in

  $$(\delta h_1, h_2.(\lambda x.h_1\langle x\rangle)h_2\langle\rangle)\lceil V_1, V_2\rceil \to_{beta} (\delta h_1, h_2.h_1\langle h_2\langle\rangle\rangle)\lceil V_1, V_2\rceil$$

  is the literal replacement of the left-hand side by the right-hand side of $(beta)$.

- Furthermore, it is assumed that the steps are closed under meta-contexts D of λc: if $U_1 \to_\beta U_2$ then $D[U_1] \to_\beta D[U_2]$.

[2] With the translation function on λ-terms being the identity function (see Definition 4.1.9), we could have said 'let U_1 and U_2 be two λ-terms.'

We emphasise that this is an alternative definition of the rewrite relation β in λc. The rewrite relation β in λc (see Definition 3.1.5) is defined in the standard way as in lambda calculus, via the rewrite rule schema (β) now over λc-terms. Note that the definition via (*beta*) does not use a rewrite rule *schema* but a rule.

Here Λ, $\langle\,\rangle$, δ and $\lceil\,\rceil$ are a part of the context machinery, which takes care of, among other things, intended variable capturing.

Formulation of β-reduction in PRSs We recall Examples 1.3.3 and 1.3.10, where lambda calculus in the PRS format has been given and repeat the pattern rewrite rule beta:

$$z_1, z_2.\, \mathsf{app}\,(\mathsf{abs}(x.\, z_1 x))\; z_2 \to\; z_1, z_2.\, z_1 z_2\;. \qquad (\mathsf{beta})$$

Here, the abstraction $\_.\_$ and (implicit) application $\_\cdot\_$ are part of the substitution calculus, which among other things, internalises intended variable capturing. The definition of the rewrite relation β on representations of λ-terms within this PRS is basically the same as the definition of β using (*beta*).

Higher-order rewriting technique for communication in λc In this comparison the role of the context-related machinery amounts to the role of the substitution calculus in PRSs. Hence, the context-related machinery uses the same techniques for dealing with intended bindings:

- In the definition of β-reduction, the hole variables are, as in the case of variables z_1 and z_2, *substituted* by representations of λ-terms with a 'communication' prefix. In contrast, in lambda calculus, M and N are literally replaced by λ-terms, and this replacement (grafting) is a meta-operation.
- The hole variables are, like z_1 and z_2, functional variables.
- The hole variables are replaced by representations of communicating λ-terms, which have a communication prefix. The communication prefix, in general $\Lambda\vec{x}.$, has the same function as the abstractor $\vec{x}.$ of the substitution calculus in PRSs: together with the arguments of the hole variables, this prefix establishes the intended bindings.

Remark 3.5.1 In fact, this usage of higher-order rewrite techniques for formalisation of communication can be extended to the usage of higher-order rewriting for formalisation of contexts in its entirety. As indicated above, a hole of a context can be represented as $z\vec{N}$ where z is understood as a hole variable and where $\vec{N}$ keep track of the relevant α, β-changes in (the representation of) the context. Communicating terms can be represented as $\vec{x}.\, M$, where $\vec{x}$ denotes the variables that are intended to become bound by the binders of a context. Then, upon substituting holes by communicating terms, communication is computed by β-reduction: viz. $(\vec{x}.\, M)\vec{N} \to_\beta M[\![\vec{x} := \vec{N}]\!]$. Continuing this line of reasoning, a context can be represented as an abstraction over a hole variable $z.\, P$. Hole filling is then represented by $(z.\, P)(\vec{x}.\, M)$ and computed by β-reduction. In sum, the substitution calculus

of a pattern rewrite system can be used for formalisation of contexts of the object language.

This kind of context formalisation using higher-order rewrite systems can be called deep embedding, because it uses the substitution calculus, which is the standard machinery of higher-order rewrite systems.

In contrast, our approach could be called shallow embedding, because the context-related issues are on the level of the object language, and hence, independent of the substitution calculus. We explain this in some detail. Note that the PRS $\mathcal{H}_c$ can be seen as a rewrite system consisting of the following levels:

- the level of the substitution calculus, that is, the level of $\lambda^{\rightarrow}_{\overline{\eta}}$-Church with function symbols ranging over $\mathcal{C}_c$;
- the level of the object language, consisting of
  - the level of term rewrite system; and,
  - the level of context-related machinery.

Here, the distinction between the substitution calculus and the object language is natural in higher-order rewriting, and the distinction between the level of term rewriting and the level of context rewriting is based on disjoint syntax. In this layered perspective on $\mathcal{H}_c$, one sees that the context-related rewriting is a part of the object language.

The definite advantage of shallow embedding is that one does not have to use the format of higher-order rewriting, and moreover, that the context-related transformations can be studied as rewrite relations, separately from the rewriting of the substitution calculus.

Remark 3.5.2 In the format of higher-order rewrite systems, our way of formalising contexts can be compared to the formalisation of contexts by D. Sands in [San98]. The similarity between the two systems is that they use the same mechanism for capturing communication: the higher-order rewriting style of capturing bindings, as described above.

However, the two formalisations use different parts of the system to capture communication. In the format of higher-order rewrite systems and in the terminology of the previous remark, our approach is an example of shallow embedding of contexts, whereas Sands' approach is an example of deep embedding. In the latter approach, contexts are formalised by (an adapted form of) the substitution calculus. More precisely, the system consists of two levels: the level of the substitution calculus and the level of the object language. The substitution calculus is a lambda-calculus-like language with multiple abstraction and multiple application. Moreover, multiple applications have a restricted form, namely, only $z\vec{M}$ where z is a meta-variable and $\vec{M}$ are terms[3]. In this system, communication is represented

[3]This is also a natural form of applications in higher-order rewriting, where one works on $\beta\overline{\eta}$-normal forms. So, here, in effect, D. Sands defines normal forms of the substitution calculus. Note that the normal forms of the substitution calculus are with respect to a multiple version of the β-rewrite rule.

by the substitution calculus.

There is also another difference between Sands' system and our calculus, which we have already indicated in Section 2.4 and Table 2.1. In Sands' system, hole filling and composition are represented by (meta-)substitution of variables, followed[4] by a reduction to β-normal form. Hence, in Sands's system hole filling and composition are meta-operations. In contrast, in our approach, hole filling and composition are rewrite relations.

3.5.2 The calculi of explicit substitutions

In the context calculus λc, communication is formalised using the higher-order rewriting technique for dealing with bindings, as discussed in the previous subsection. Using this technique, essentially the substitution σ at a hole $[]^\sigma$ that arises from rewriting in a context is encoded. Thus, in this sense, the context calculus encodes substitutions. A natural question is then how the context calculus compares to the calculi of explicit substitutions.

Actually, this link is a bit misleading. There are two ways to compare the context calculus and the calculi of explicit substitutions, but both ways illustrate a different approach to substitutions. We describe and discuss these comparisons in turn.

The first comparison is a technical one. In the context calculus, substitutions can be represented but their computation is not formalised in a stepwise manner as in the calculi of explicit substitutions. More precisely, a substitution $[\![\vec{x} := \vec{N}]\!]$ to be applied to M can be represented as $(\Lambda\vec{x}.\, M)\langle\vec{N}\rangle$. It may be computed by the rewrite rule $(\textcircled{m}\beta)$:

$$(\Lambda\vec{x}.\, M)\langle\vec{N}\rangle \to M\,[\![\vec{x} := \vec{N}]\!]\,.$$

What is important here is that computation of this substitution is carried out by a meta-operation, which applies $[\![\vec{x} := \vec{N}]\!]$ instantaneously to M.

This is in contrast to the calculi of explicit substitutions. There, substitutions can be represented, and their computation is formalised in a stepwise manner: a substitution like $[\![\vec{x} := \vec{N}]\!]$ traverses through a term M, following the structure of M step by step.

The second comparison is a conceptual one. The context calculus encodes the substitutions that arise from rewriting in contexts. The computation of these substitutions are encoded in a two-step manner. Suppose $[\![x := N]\!]$ is the (meta-)substitution that arose from a rewrite step in a context and consider a hole of the context. In the context calculus, if the hole lies in the scope of the substitution, the substitution is applied to (the representation of) the hole $h\langle\vec{x}\rangle$, resulting in $h\langle x_1, \ldots, N, \ldots, x_n\rangle$. Conceptually, this is the first substitution step, and it is

[4] After substitution, this reduction is necessary in order to maintain the restricted form of multiple applications, or, in other words, in order to maintain working on normal forms with respect to the multiple version of the β-rewrite rule.

carried out by the same meta-operation that arose from the rewrite step in the context.

The second substitution step arises upon filling the hole by a communicating term $\Lambda\vec{x}.\, M$: a communication redex is formed, which reconstructs the original substitution. Contracting this communication redex employs again a meta-operation: the (meta-)substitution $[\![\vec{x} := \vec{N}]\!]$ is applied to M instantaneously.

Again, this is in contrast to the calculi of explicit substitutions.

Chapter 4

Applications of λc

The framework λc can be used for representing different notions of context in the untyped and simply typed lambda calculus. A particular variation can be obtained by fine-tuning typing. This flexibility of the framework λc will be illustrated by four examples: three examples in this chapter and an extended example in the next chapter. Each example represents contexts by applying the method that was described in Section 2.5. The main differences between the examples lie in the notion of context and the presence of variables and functions ranging over contexts.

These examples are defined via a collection of typing rules. As already explained in Section 2.5, a particular collection of typing rules defines a set of well-typed λc-terms, as a subset of all λc-terms. Indirectly, a particular collection of typing rules also fixes a subset of the rewrite rules of λc, namely the subset of the rewrite rules which are applicable to the well-typed λc-terms. In sum, a collection of typing rules determines a subset of λc-terms and a subset of the rewrite rules of λc.

The questions that arise regarding the properties of type systems are of two kinds. The first kind of questions is related to whether a particular typing defines a subsystem (in the sense of Definition 1.1.14) of the framework λc, which is then closed under rewriting. The second kind of questions is related to investigating the properties of rewriting, in particular the confluence and normalisation properties.

This chapter is organised as follows.

In Section 4.1 the calculus λc^{λ} is defined, which is a calculus for representing the untyped lambda calculus with λ-contexts. In this calculus, representations of λ-contexts can be manipulated, but there are no variables or functions ranging over (representations of) λ-contexts. That is, the calculus λc^{λ} does not have a fully first-class treatment of λ-contexts. This calculus is basically lambda calculus with an adequate notation for contexts, on which now also the β-rewrite relation is defined. The calculus λc^{λ} has the confluence property and the context-related rewriting is strongly normalising.

In Section 4.2 the calculus $\lambda c^{\rightarrow}$ is defined, which is a calculus over simply typed lambda calculus with λ-contexts. This calculus includes variables and functions ranging over contexts. It represents the most natural notion of λ-context, and it

is comparable to the context calculus of Hashimoto and Ohori (see [HO98]). The calculus $\lambda c^{\rightarrow}$ is complete, that is, it has the confluence property and the strong normalisation property.

In Section 4.3 the calculus $\lambda c^{\cong}$ is defined, which is a calculus over untyped lambda calculus with λ-contexts as first-class citizens. This calculus is interesting for two reasons. The first reason is that this calculus describes the minimal conditions which guarantee the well-formedness of the context machinery. Note that $\lambda c^{\cong}$ deals with representations of *untyped* λ-terms and *untyped* λ-contexts; the typing pertains only to the issues dealing with representation of such terms and contexts and with representation of context-related (meta-)operations. The second reason is that both λc^{λ} and $\lambda c^{\rightarrow}$ can be translated into $\lambda c^{\cong}$. This calculus is a variation of the calculus $\lambda c^{\rightarrow}$ obtained by ignoring the types (of the simply typed lambda calculus). It is also an extension of the calculus λc^{λ} with first-class λ-contexts. The calculus $\lambda c^{\cong}$ has the confluence property and the rewriting dealing with communication, hole filling and functions ranging over contexts (including composition) is strongly normalising.

In all examples, the proofs of the properties studied resemble each other. In order to be short in the succeeding examples, we will write out the proofs of these properties in the first example.

In Section 4.4 we summarise the relationship between the systems considered in this chapter.

4.1 The calculus λc^{λ}

The context calculus λc^{λ} formalises the untyped lambda calculus with λ-contexts. In λc^{λ}, the untyped λ-contexts and context-related (meta-)operations can be represented. However, in λc^{λ} there are no variables or abstractions over (representations of) contexts. That is, the calculus λc^{λ} formalises contexts in the way they are used in lambda calculus, only now in λc^{λ} the standard lambda calculus transformations are defined on (representations of) contexts.

This section is split into seven subsections:

- Lambda objects: We will first recall the well-known definitions of λ-terms and transformations on λ-terms, λ-contexts and context-related operations, as given in the introduction. Using these definitions we will define the lambda objects and the transformations we wish to represent within λc.

- Translation of lambda objects into λc: Representation of lambda objects within λc follows the approach described in Section 2.5. We make it precise by defining a translation function from lambda objects to λc.

- The calculus λc^{λ}: Keeping in mind the expressions we wish to represent (i.e. lambda objects) and the way we want to represent them (i.e. translation), we switch to the framework λc. We give a set of typing rules and a collection of rewrite rules for λc^{λ}. The rewrite rules are the rewrite rules of the framework λc. Typing in λc^{λ} pertains to controlling the context-related machinery, and

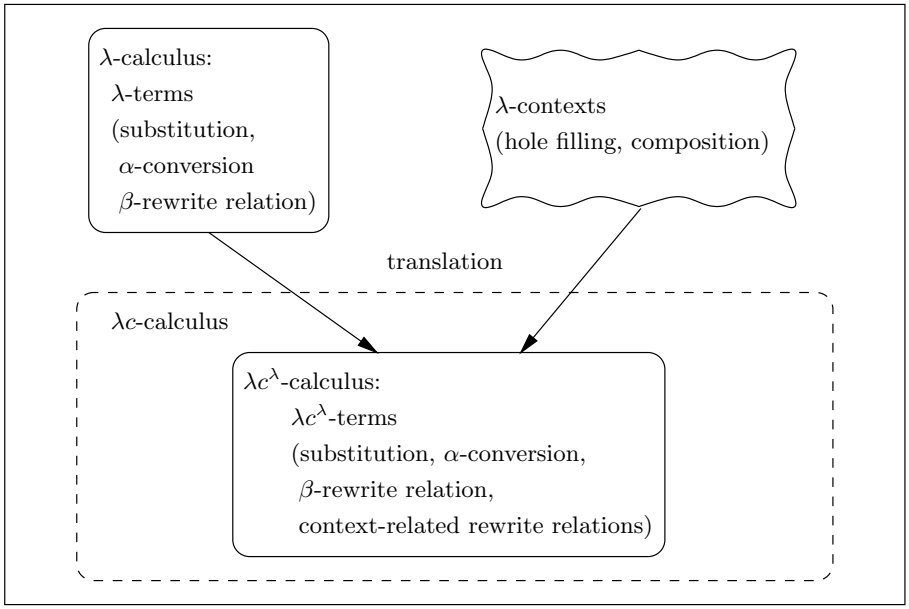


Figure 4.1: Contents of Section 4.1

not to the functionality of terms as in for example the simply typed lambda calculus. In this sense, typing in λc^{λ} can be called syntactic typing.

- The calculus λc^{λ} is a subsystem of the framework λc: We prove that the set of well-typed terms of λc^{λ} is closed under rewriting of λc^{λ}.

- Properties of the subsystem λc^{λ}: We show that λc^{λ} has the confluence property and that the context-related rewriting has the strong normalisation property. The strong normalisation property of the context-related rewriting is proved via a computability predicate on terms, as in Tait's well-known method.

- Adequacy of context representation in λc^{λ}: The question in this subsection is how adequate the calculus λc^{λ} is for representing the lambda calculus contexts. To answer this question, we use the translation of the lambda objects into the calculus λc^{λ} and show that λc^{λ} meets our goals: the specified lambda objects can be represented within λc^{λ}, and in λc^{λ} the standard lambda calculus transformations are defined on (the representations of) the lambda objects.

- The introductory example: we show how the hole filling of Example 2.5.3 can be represented within λc^{λ}.

The relationship between the formal systems involved is sketched in Figure 4.1.

Lambda objects

The objects of lambda calculus that will be represented within λc^{λ} include the untyped λ-terms and λ-contexts. The λ-terms with substitution, α-conversion and β-rewrite relation, and different notions of λ-context have already been described in Section 2.1. Here we repeat the definition of λ-terms and single out the notion of λ-context we wish to represent, namely the contexts of Definition 2.1.3 (which was also given in the preliminary chapter as Definition 1.2.5). In addition to λ-terms and λ-contexts, lambda objects include more complex objects in which (the meta-operations) hole filling and composition may occur. Complex lambda objects will shortly be explained into more detail; we recall the definitions of λ-terms, λ-contexts, hole filling and composition first.

Definition 4.1.1 (λ-terms and λ-contexts (Definitions 1.2.1 and 2.1.3))

i) The λ-terms are defined as:

$$M \quad ::= \quad x \mid \lambda x.\, M \mid M\, N.$$

ii) The λ-contexts are λ-terms with holes, that is,

$$C \quad ::= \quad x \mid [] \mid \lambda x.\, C \mid C\, D.$$

We consider each occurrence of a box [] as a different hole and assume the holes in a λ-context to be ordered from left to right. Substitution, α-conversion and β-rewrite relation are defined only on λ-terms as in Definitions 1.2.8 and 1.2.11.

Definition 4.1.2 (Hole filling, composition (Definition 2.1.3))

i) Hole filling involves a λ-context C with n holes and n λ-terms $\vec{M}$, and results in the λ-term $C[\vec{M}]$, which is the result of replacing the i^{th} hole in C by M_i (for $1 \leq i \leq n$).

ii) Composition involves a λ-context C with n holes and n λ-contexts $\vec{D}$, and results in the λ-context $C[\vec{D}]$, which is the result of replacing the i^{th} hole in C by D_i (for $1 \leq i \leq n$).

The choice for this notion of context, in particular, the choice for the number of allowed []'s and the treatment of []'s has already been discussed in Section 2.2. Many []'s are allowed because this form is invariant under β-rewriting. Because []'s may occur in the scopes of different binders in a context, and therefore filling the same term in different []'s may lead to different variable capturings in (the copies of) the term, all []'s are considered as different; hence they stand for occurrences of different holes.

In lambda calculus the result of applying both of these meta-operations is denoted by using the square brackets [ ]. In this section, we will formalise the well-known informal notion of [ ], and we will distinguish between the notations for the

meta-operations (by introducing the function symbols *hf* and *comp*) and the notations for the result of applying the meta-operations (by using []). Such a distinction enables us to postpone the evaluation of a meta-operation.

Complex lambda objects can now be obtained from λ-terms and λ-contexts by using the function symbols *hf* and *comp*. The definition of all lambda objects that we consider and the definition of the transformations on lambda objects are given below. In the definition of lambda objects it is assumed that the number of arguments $\vec{M}$ and $\vec{P}$ in respectively $hf(P, \vec{M})$ and $comp(P, \vec{P})$ matches the number of holes in the result of evaluating P. For this reason, lambda objects P are defined together with the evaluation function P^*, which evaluates hole filling and composition within the lambda object P.

Definition 4.1.3 (Lambda objects) Lambda objects P and the evaluation function P^* are defined simultaneously as follows:

i) Terms M and contexts C as defined in Definition 4.1.1 are lambda objects, and

$$\begin{array}{lcl} M^* & = & M \\ C^* & = & C. \end{array}$$

ii) Let P be a lambda object that evaluates to a context with n holes, i.e. P^* is a context with n holes.

Let $\vec{R}$ be n lambda objects that evaluate to a term, i.e. R_i^* is a term for each $1 \leq i \leq n$.

Then $hf(P, \vec{R})$ is a lambda object and $(hf(P, \vec{R}))^* = P^*[R_1^*, \ldots, R_n^*]$.

iii) Let P be a lambda object that evaluates to a context with n holes, i.e. P^* is a context with n holes.

Let $\vec{P}$ be n lambda objects that evaluate to a context, i.e. P_i^* is a context for each $1 \leq i \leq n$.

Then $comp(P, \vec{P})$ is a lambda object and $(comp(P, \vec{P}))^* = P^*[P_1^*, \ldots, P_n^*]$.

Remark 4.1.4 Assuming that the number of arguments $\vec{R}$ and $\vec{P}$ in respectively $hf(P, \vec{R})$ and $comp(P, \vec{P})$ matches the number of holes in the result of evaluating P, the lambda objects can be defined as:

$$\begin{array}{lcl} R & ::= & M \mid hf(P, \vec{R}) \\ P & ::= & C \mid comp(P, \vec{P}). \end{array}$$

Definition 4.1.5 (Transformations on lambda objects) The substitution, α-conversion and β-rewrite relation is defined as in Section 1.2.1 only on the λ-terms M (see Definition 4.1.1).

Example 4.1.6 An example of a lambda object is $hf(comp(\lambda x.[],\ [][]),\ x, y)$, which evaluates to $\lambda x.\, x\, y$. Another example is $hf(x)$. Here, x is a context without holes, which is allowed by Definition 4.1.1; hence hf has no other arguments. This lambda object evaluates to x, i.e. $(hf(x))^* = x$.

In informal lambda calculus, where there is no distinction between denoting the meta-operation $[\,]$ and its result, the formation of λ-terms or λ-contexts can be combined with filling holes. For example $\lambda x.\, C[M]$ is a λ-term, where M is a λ-term and C a λ-context. Here, it is not clear whether $\lambda x.\, C[M]$ is to be considered as the result of $hf((\lambda x.[]),\ hf(C,\ M))$ or $hf(comp((\lambda x.[]),\ C),\ M)$. In contrast, with our explicit notation using hf and $comp$, this ambiguity is automatically cleared up.

Notation. In the sequel $P, \vec{P}, \vec{R}$ range over expressions possibly containing hf and $comp$. Here P and $\vec{P}$ will be used for expressions which evaluate to a λ-context, and $\vec{R}$ for expressions resulting in λ-terms, unless explicitly stated otherwise.

Translation of lambda objects into λc

Translation of the λ-contexts and the context-related functions to λc requires some preprocessing, which involves meta-level (α-sensitive) observations being made explicit. For this purpose, two functions are assumed:

- $\textsc{Nrh}(P)$, which returns the number of holes in the result of evaluating P (i.e. in P^*); and
- $\textsc{Bnd}(P, i)$, which returns the list of all variables such that the i^{th} hole of P^* lies in their scope ($1 \leq i \leq \textsc{Nrh}(P)$); the variables are ordered by left–to–right appearance in the term.

For instance, $\textsc{Nrh}(comp(\lambda x.[],\ [][])) = 2$ and $\textsc{Bnd}(comp(\lambda x.[],\ [][]), 1) = x$. Regarding Bnd, we assume that there are no 'overshadowed' binders in a λ-context, like the leftmost λx in $\lambda x.\, \lambda x.\, []$, but Bnd (and later, the λc-terms $L^{\vec{x}}_P$) could have been defined otherwise to deal with 'overshadowed' variables.

Remark 4.1.7 In a λ-context C, we can choose another name for an 'overshadowed' binder, because that binder will not capture any free variables of λ-terms or λ-contexts that are put into the holes of C. For example, the leftmost λx can be renamed into λy, viz. $\lambda x.\, \lambda x.\, [] = \lambda y.\, \lambda x.\, []$, if and only if y is not free in terms $\vec{M}$ or contexts $\vec{D}$ to be put into the holes of this context. See also Remark 1.2.10.

The λ-terms and λ-contexts are translated to λc by the translation function $[\![\_]\!]$. The translation function behaves like the identity function on λ-terms while on λ-contexts it replaces holes by 'labelled' hole variables and adds a preamble. Translation of an arbitrary expression explicitly involves hole filling, composition, communication and lifting of holes. Translation will first be illustrated by an extended example, and then defined formally.

Example 4.1.8 We consider one λ-term M and two λ-contexts C and D, defined below, and explain the translations of M, C, D, $hf(C,\ M)$ and $comp(C,\ D)$. This example is split into two parts, the easy one dealing with the translations of M and C; and the difficult one, dealing with the translations of $hf(C,\ M)$ and $comp(C,\ D)$. The translations follow the method described in Section 2.5.

Let

$$\begin{array}{lcl} M & \equiv & x\,y \\ C & \equiv & (\lambda y.\,[])\,x \\ D & \equiv & []\,(\lambda z.\,[]). \end{array}$$

Then,

$$\begin{array}{l} \textsc{Nrh}(C) = 1, \quad \textsc{Bnd}(C,1) = y, \\ \textsc{Nrh}(D) = 2, \quad \textsc{Bnd}(D,1) = \epsilon \quad \text{and} \quad \textsc{Bnd}(D,2) = z. \end{array}$$

i) Translation of λ-terms is trivial, because λ-terms are also λc-terms:

$$\llbracket M \rrbracket = M = x\,y.$$

Translation of a context replaces the i^{th} hole by $h_i\,\langle \vec{x} \rangle$ where $\vec{x}$ are the variables in whose scope the i^{th} hole lies, and adds a multiple abstraction $\delta\vec{h}$ over the hole variables as a preamble:

$$\begin{array}{lcllcl} \llbracket C \rrbracket & = & \delta g.\, C[g\,\langle y \rangle] & = & \delta g.\,(\lambda y.\, g\,\langle y \rangle)\,x \\ \llbracket D \rrbracket & = & \delta k_1, k_2.\, D[k_1\,\langle\rangle, k_2\,\langle z \rangle] & = & \delta k_1, k_2.\,(k_1\,\langle\rangle)\,(\lambda z.\, k_2\,\langle z \rangle). \end{array}$$

Note that λ-contexts are also meta-contexts of λc. In the equations above, the λc-term $C[g\,\langle y \rangle]$ denotes the result of the meta-operation of filling the hole of C, which is here considered as a meta-context of λc. An analogous statement holds for $D[k_1\,\langle\rangle, k_2\,\langle z \rangle]$.

ii) The translation of the hole filling $hf(C,\ M)$ proceeds as follows. The hole $[]$ of C is in the scope of the abstractor λy. Because the translation of M is to be put into the hole of the translation of C, the communication preamble Λy is added to the translation of M. This results in the communicating λc-term

$$\Lambda y.\, \llbracket M \rrbracket = \Lambda y.\, x\,y.$$

The translation of the hole filling $hf(C,\ M)$ is the hole filling of λc applied to the translation of C and the communicating term:

$$\llbracket\ hf(C,\ M)\ \rrbracket = \llbracket C \rrbracket\,\lceil \Lambda y.\, \llbracket M \rrbracket \rceil = (\delta g.\,(\lambda y.\, g\,\langle y \rangle)\,x)\,\lceil \Lambda y.\, x\,y \rceil.$$

One may check that this term reduces to $(\lambda y.\, x\,y)\,x$, which is the translation of the λ-term $C[M]$.

The translation of the composition $comp(C,\ D)$ is analogous to the translation of the hole filling, but in addition it involves lifting of the holes of the context D. It proceeds as follows. Because the translation of D is to be put into the hole of the translation of C, the communication preamble Λy is added to the

translation of D. In addition, the holes of the translation of D are lifted by the variables of the communication preamble. The lifting is done explicitly in the λc-term L^y_D, which is specifically designed for D and y. This results in the communicating λc-term

$$\Lambda y.\, L^y_D$$

where (recall that $\textsc{Bnd}(D,1) = \epsilon$ and $\textsc{Bnd}(D,2) = z$)

$$L^y_D = \delta h_1, h_2.\, \llbracket D \rrbracket\, \lceil \Lambda\epsilon.\, h_1\, \langle y \rangle, \Lambda z.\, h_2\, \langle y, z \rangle \rceil.$$

One may check that the communicating term $\Lambda y.\, L^y_D$ reduces in λc^λ to $\Lambda y.\, \delta h_1, h_2.\, h_1\, \langle y \rangle\, (\lambda z.\, h_2\, \langle y, z \rangle)$, which is the translation of the λ-context D with lifted holes and a communication preamble. The translation of the composition $\mathit{comp}(C,\, D)$ is the composition of λc applied to the translation of C and the communicating term ($\circ$ is used in infix notation because in this example it is binary):

$$\llbracket\ \mathit{comp}(C,\, D)\ \rrbracket \quad = \quad \llbracket C \rrbracket \circ (\Lambda y.\, L^y_D).$$

One may check that this term reduces to $\delta h_1, h_2.\, (\lambda y.\, h_1\, \langle y \rangle\, (\lambda z.\, h_2\, \langle y, z \rangle))\, x$, which is the translation of $C[D]$.

Definition 4.1.9 (Translation of lambda objects into λc)

i) Let M be a λ-term and C a λ-context. Let $\textsc{Nrh}(C) = n$, $\textsc{Bnd}(C,1) = \vec{x}_1$, ...,$\textsc{Bnd}(C,n) = \vec{x}_n$. Then

$$\begin{array}{lcl} \llbracket M \rrbracket & = & M \\ \llbracket C \rrbracket & = & \delta \vec{h}.\, C[h_1\, \langle \vec{x}_1 \rangle, \ldots, h_n\, \langle \vec{x}_n \rangle]. \end{array}$$

ii) The translation function extends to the composed objects as: if P and $\vec{P}$ are expressions evaluating to λ-contexts with $\textsc{Nrh}(P) = n$, and $\textsc{Bnd}(P,1) = \vec{x}_1, \ldots, \textsc{Bnd}(P,n) = \vec{x}_n$, and if $\vec{R}$ are expressions evaluating to λ-terms, then

$$\begin{array}{lcl} \llbracket\ \mathit{hf}(P,\, \vec{R})\ \rrbracket & = & \llbracket P \rrbracket\, \lceil \Lambda \vec{x}_1.\, \llbracket R_1 \rrbracket, \ldots, \Lambda \vec{x}_n.\, \llbracket R_n \rrbracket \rceil \\ \llbracket\ \mathit{comp}(P,\, \vec{P})\ \rrbracket & = & \llbracket P \rrbracket \circ (\Lambda \vec{x}_1.\, L^{\vec{x}_1}_{P_1}, \ldots, \Lambda \vec{x}_n.\, L^{\vec{x}_n}_{P_n}), \end{array}$$

where $L^{\vec{x}_i}_{P_i}$ lifts the holes of $\llbracket P_i \rrbracket$. In general, the term $L^{\vec{x}_i}_{P_i}$ is defined by: if $\textsc{Nrh}(P_i) = m$, $\textsc{Bnd}(P_i,1) = \vec{y}_1, \ldots, \textsc{Bnd}(P_i,m) = \vec{y}_m$, then

$$L^{\vec{x}_i}_{P_i} = \delta \vec{g}.\, \llbracket P_i \rrbracket\, \lceil \Lambda \vec{y}_1.\, g_1\, \langle \vec{x} \vec{y}_1 \rangle, \ldots, \Lambda \vec{y}_m.\, g_m\, \langle \vec{x} \vec{y}_m \rangle \rceil.$$

Remark 4.1.10 Note that translation involves explicit lifting of holes. An alternative would be to parametrise the translation of λ-contexts by the lifting variables. We explain this into more detail. Let D be the λ-context as in Example 4.1.8 and let D' be the communicating λ-context

$$D' \;=\; D \text{ where } y \text{ will become bound after composition with } C.$$

We could, for example, define the translation of D already parametrised by the communication y as $[\![D]\!]_y = \delta k_1, k_2.\, D[k_1\langle y\rangle, k_2\langle y, z\rangle]$. The translation of D' would in that case only add the preamble Λy, that is, $[\![D']\!] = \Lambda y.\, [\![D]\!]_y$.

We prefer the explicit lifting because in this approach the translation $[\![D]\!]$ is independent from the communication with C. That is, the translation of the communicating λ-context D' can be seen as a function F of the translation of D and the translation of the communication

$$[\![D']\!] = F([\![D]\!], y) = \Lambda y.\, \delta h_1, h_2.\, [\![D]\!]\, \lceil \Lambda \epsilon.\, h_1\langle y\rangle, \Lambda z.\, h_2\langle y, z\rangle \rceil.$$

Example 4.1.11 We give an example of the translation of a lambda object with nested meta-operations. Let M,C and D be as in Example 4.1.8. Consider the lambda object $hf(comp(C,\, D),\, M, M)$. Because $comp(C,\, D)$ evaluates to a context with two holes, the hole filling hf has two additional arguments; in this example M is filled in both holes. The translation of $hf(comp(C,\, D),\, M, M)$ is

$$[\![\, hf((comp(C,\, D)),\, M, M)\,]\!] = ([\![C]\!] \circ (\Lambda y.\, L^{y}_{D}))\, \lceil \Lambda y.\, [\![M]\!], \Lambda y, z.\, [\![M]\!] \rceil.$$

Note that the communication preambles of the two translations of M are different. Note also that Bnd and Nrh (by definition) operate on evaluated lambda objects.

The calculus λc^{λ}

For the time being, we keep the lambda objects and their translation in the background, switch to the context calculus, and define a subsystem of λc that will correspond to lambda objects as close as possible. We start with the definition of types, continue with the type system (Definition 4.1.13) and end with the definition of λc^{λ} (Definition 4.1.14).

As already remarked, the typing of λc^{λ} may be called syntactic because its role is only to control the well-formedness of the context-related machinery of the calculus. For example, typing ensures that in a communication redex the number of arguments of the multiple application and the number of variables in the multiple abstraction agree, that hole filling provides the right number of arguments etc. This kind of typing does not restrict the formation of the terms of the untyped lambda calculus.

Definition 4.1.12 (Types of λc^{λ}) Let the set of base types be $\mathcal{V}^{\lambda} = \{\mathtt{t}\}$. Then the types $\rho \in \mathcal{P}_{\lambda}$ are defined as

$$\rho ::= \mathtt{t} \;\mid\; [\vec{\mathtt{t}}]\mathtt{t} \;\mid\; [\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t} \;\mid\; [\vec{\mathtt{t}}]([\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}),$$

where $[\,]$ binds stronger than $\times$, and $\times$ binds stronger than $\Rightarrow$.

The new type constructors $[\_,\ldots,\_]\_$ and $\_\times\ldots\times\_\Rightarrow$ are introduced for distinguishing between different pairs of an abstractor and an applicator of λc, namely, $[\_,\ldots,\_]\_$ for the pair Λ, $\langle\,\rangle$, and $\_\times\ldots\times\_\Rightarrow$ for the pair δ, $\lceil\,\rceil$, as will become clear in the typing rules. Both type constructors are comparable to $\_\to\ldots\to\_\to\_$ in $\lambda^{\to}$. Due to the correspondence between these new type constructors and λc-term constructors, an intuition can be given about the types (this intuition applies also to the calculi $\lambda c^{\to}$ and $\lambda c^{\cong}$ defined later in this chapter):

- the type $\mathtt{t}$ will be the type of representations of λ-terms, for example

$$(xy)\ :\ \mathtt{t};$$

- a type of the form $[\vec{\mathtt{t}}_1]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_n]\mathtt{t}\Rightarrow\mathtt{t}$ will be the type of the representations of λ-contexts with n holes, for example

$$(\delta h_1,h_2.(h_1\,\langle\rangle)(\lambda z.h_2\,\langle z\rangle))\ :\ [\,]\,\mathtt{t}\times[\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t};$$

- types of the form $[\vec{\mathtt{t}}]\mathtt{t}$ and $[\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\,\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_n]\mathtt{t}\Rightarrow\mathtt{t})$ will be the types of the representations of communicating λ-terms and λ-contexts, respectively, with the types of objects involved in communication sequenced between the square brackets, for example

$$(\Lambda y,z.xy)\ :\ [\mathtt{t},\mathtt{t}]\mathtt{t}$$
$$(\Lambda y.\delta h_1,h_2.(h_1\,\langle y\rangle)(\lambda z.h_2\,\langle y,z\rangle))\ :\ [\mathtt{t}]([\mathtt{t}]\,\mathtt{t}\times[\mathtt{t},\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t}).$$

The typing rules use two bases: the basis Γ, which is a set of declarations of the form $x:\mathtt{t}$; and the basis Σ, which is a set of declarations of the form $h:[\vec{\mathtt{t}}]\mathtt{t}$. The bases are split because the elements of Γ are used as true variables (i.e. placeholders for representations of λ-terms) whereas the elements of Σ serve as markers, in the sense that they are used for marking the beginning (abstraction) and endings (i.e. holes) of a context. The 'marker function' of hole variables can be seen in the typing rules (see Figure 4.1.13) by the positions where hole variables may (not) occur. For example, hole variables cannot occur among $x,\vec{x}$ in the abstractions $\lambda x.U$ and $\Lambda\vec{x}.U$, where $x,\vec{x}$ stand for (representations of) λ-terms.

In the typing rules, $\vec{U}:\vec{\mathtt{t}}$ denotes the pointwise typing $U_i:\mathtt{t}$ for $1\leq i\leq|\vec{U}|=|\vec{\mathtt{t}}|$. Furthermore, both Γ and Σ are, without loss of generality, assumed to contain distinct variables.

Definition 4.1.13 (Type system for λc^{λ}) A term $U\in\textsc{Ter}(\lambda c)$ is typable by ρ from the bases Γ,Σ, if $\Gamma;\Sigma\vdash U:\rho$ can be derived using the typing rules displayed in Figure 4.2.

The typing rules can be explained as follows. The rules (var), (abs) and (app) are the rules that guard the well-formedness of the untyped λ-terms. In these typing rules all terms and variables are typed by $\mathtt{t}$. The rules $(hvar)$ and $(habs)$ are used

$$(var)\quad \frac{(x : \mathtt{t}) \in \Gamma}{\Gamma; \Sigma \vdash x : \mathtt{t}}$$

$$(abs)\quad \frac{\Gamma, x : \mathtt{t}; \Sigma \vdash U : \mathtt{t}}{\Gamma; \Sigma \vdash \lambda x.U : \mathtt{t}}$$

$$(app)\quad \frac{\Gamma; \Sigma \vdash U : \mathtt{t} \qquad \Gamma; \Sigma \vdash V : \mathtt{t}}{\Gamma; \Sigma \vdash UV : \mathtt{t}}$$

$$(hvar)\quad \frac{(h : [\vec{\mathtt{t}}]\mathtt{t}) \in \Sigma}{\Gamma; \Sigma \vdash h : [\vec{\mathtt{t}}]\mathtt{t}}$$

$$(mabs)\quad \frac{\Gamma, \vec{x} : \vec{\mathtt{t}}; \Sigma \vdash U : \mathtt{t}}{\Gamma; \Sigma \vdash \Lambda\vec{x}.U : [\vec{\mathtt{t}}]\mathtt{t}}$$

$$(mapp)\quad \frac{\Gamma; \Sigma \vdash U : [\vec{\mathtt{t}}]\mathtt{t} \qquad \Gamma; \Sigma \vdash \vec{V} : \vec{\mathtt{t}}}{\Gamma; \Sigma \vdash U\langle\vec{V}\rangle : \mathtt{t}}$$

$$(habs)\quad \frac{\Gamma; \Sigma, h_1 : [\vec{\mathtt{t}}_1]\mathtt{t}, \ldots, h_n : [\vec{\mathtt{t}}_n]\mathtt{t} \vdash U : \mathtt{t}}{\Gamma; \Sigma \vdash \delta\vec{h}.U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}}$$

$$(fill)\quad \frac{\begin{array}{l}\Gamma; \Sigma \vdash U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} \\ \Gamma; \Sigma \vdash V_i : [\vec{\mathtt{t}}_i]\mathtt{t} \qquad (1 \leq i \leq n)\end{array}}{\Gamma; \Sigma \vdash U\lceil\vec{V}\rceil : \mathtt{t}}$$

$$(mabs_c)\quad \frac{\Gamma, \vec{x} : \vec{\mathtt{t}}; \Sigma \vdash U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}}{\Gamma; \Sigma \vdash \Lambda\vec{x}.U : [\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t})}$$

$$(comp)\quad \frac{\begin{array}{l}\Gamma; \Sigma \vdash U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} \\ \Gamma; \Sigma \vdash V_i : [\vec{\mathtt{t}}_i]([\vec{\mathtt{t}}_{i,1}]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{i,l_i}]\mathtt{t} \Rightarrow \mathtt{t}) \quad (1 \leq i \leq n)\end{array}}{\Gamma; \Sigma \vdash U \circ \vec{V} : [\vec{\mathtt{t}}_{1,1}]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{n,l_n}]\mathtt{t} \Rightarrow \mathtt{t}}$$

Figure 4.2: Type system for λc^{λ}

in forming representations of λ-contexts. The rule (*habs*) could also be presented more transparently as an abstraction typing rule, by

$$(habs)\quad \frac{\Gamma;\Sigma, h_1:\rho_1,\ldots h_n:\rho_n \vdash U:\mathtt{t}}{\Gamma;\Sigma \vdash \delta\vec{h}.U:\rho_1\times\ldots\times\rho_n\Rightarrow\mathtt{t}.}$$

However, we prefer the form as given in the figure, where ρ_i's are fully specified because there is only one form of type that ρ_i's can assume.

The rules (*mabs*), ($mabs_c$) and (*mapp*) are used for typing representations of holes and λ-terms and λ-contexts to be put into holes. Note that only elements of λc^{λ} of type $\mathtt{t}$ may be involved in communication. The rules (*fill*) and (*comp*) are used in typing when filling holes. The rule (*fill*) is the application counterpart of the rule (*habs*), and it could be presented as

$$(fill)\quad \frac{\Gamma;\Sigma \vdash U:\rho_1\times\ldots\times\rho_n\Rightarrow\mathtt{t}\quad \Gamma;\Sigma\vdash V_i:\rho_i}{\Gamma;\Sigma\vdash U\lceil\vec{V}\rceil:\mathtt{t}}$$

with the types $\rho_1,\ldots,\rho_n$ of a specific form. The rule (*comp*) is a rather complicated one, because the composition of λ-contexts, which is typed by this rule, is also complicated. In this rule, the term U is a representation of the outer λ-context, and V_i's are representations of λ-contexts to be filled into the holes of the outer λ-context. This rule requires that the communication of the i^{th} hole of the outer λ-context agrees with the communication of V_i (in both cases, $\vec{\mathtt{t}}_i$). The type $[\vec{\mathtt{t}}_{1,1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{n,l_n}]\mathtt{t}\Rightarrow\mathtt{t}$ of the result indicates a representation of a context over the holes of V_1 through V_n. We illustrate this rule by an example with concrete types. This is an example of the composition typing rule where the two holes of the outer context are filled by contexts with two and one holes respectively.

example of (*comp*)

$$\frac{\begin{array}{lll}\Gamma;\Sigma & \vdash & U:[\mathtt{t}]\,\mathtt{t}\times[\mathtt{t},\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t}\\ \Gamma;\Sigma & \vdash & V_1:[\mathtt{t}]([\mathtt{t}]\,\mathtt{t}\times[\mathtt{t},\mathtt{t},\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t})\\ \Gamma;\Sigma & \vdash & V_2:[\mathtt{t},\mathtt{t}]([\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t})\end{array}}{\Gamma;\Sigma\vdash U\circ(V_1,V_2):[\mathtt{t}]\,\mathtt{t}\times[\mathtt{t},\mathtt{t},\mathtt{t}]\,\mathtt{t}\times[\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t}}$$

Last but not least, note that there are no variables or abstractions over variables of type $[\vec{\mathtt{t}}_1]\,\mathtt{t}\times\ldots[\vec{\mathtt{t}}_n]\mathtt{t}\Rightarrow\mathtt{t}$, that is, context variables and functions ranging over contexts are not typable.

Like any other example of typing, the calculus λc^{λ} is defined on well-typed terms and with a rewrite relation generated by a subset of the rewrite rules of the framework λc. In a case of an arbitrary typing, the subset of rewrite rules contains all rewrite rules of the framework λc that are applicable to well-typed terms of the typing. In the case of λc^{λ}, the subset of rewrite rules consists of all rewrite rules of λc.

Definition 4.1.14 (λc^{λ}) The terms of λc^{λ} are the well-typed terms of λc according to Definition 4.1.13. The rewrite rules are the rules (β), ($@\beta$), (*fill*) and ($\circ$) of λc, now restricted to λc^{λ}-terms.

i) The lambda calculus rewrite rule is:

$$(\lambda x.U)V \to U[\![x := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$(\Lambda\vec{x}.U)\langle\vec{V}\rangle \to U[\![\vec{x} := \vec{V}]\!] \qquad (\textcircled{m}\beta)$$
$$(\delta\vec{h}.U)\lceil\vec{V}\rceil \to U[\![\vec{h} := \vec{V}]\!] \qquad (\mathit{fill})$$
$$(\delta_n\vec{g}.U) \circ (\Lambda\vec{x}_1.\delta\vec{h}_1.V_1, \ldots, \Lambda\vec{x}_n.\delta\vec{h}_n.V_n)$$
$$\to \delta\vec{h}_1, \ldots, \vec{h}_n.U[\![g_1 := \Lambda\vec{x}_1.V_1, \ldots, g_n := \Lambda\vec{x}_n.V_n]\!]. \qquad (\circ)$$

Definition 4.1.15 Let the ARS $\mathcal{A}_{\lambda c^\lambda}$ be

$$\mathcal{A}_{\lambda c^\lambda} = \langle \textsc{Ter}(\lambda c^\lambda), \to_\beta, \to_{\textcircled{m}\beta}, \to_{\mathit{fill}}, \to_\circ \rangle.$$

We call the ARS $\mathcal{A}_{\lambda c^\lambda}$ the underlying ARS of the calculus λc^λ.

The calculus λc^λ is a subsystem of the framework λc

We show that (the underlying ARS of) the calculus λc^λ is a subsystem of (the underlying ARS of) the framework λc in the sense of Definition 1.1.14. This is entailed by the closure property of the set of terms of λc^λ under rewriting. We prove that the set of terms of λc^λ is closed under substitution and rewriting, that is, we show that these transformations applied to well-typed terms result in a well-typed term. The proofs of these closure properties are conducted in a standard way (see for example [Bar92]) via the Generation, Bases thinning, Substitution and Subject reduction lemmas. First we prove the Generation lemma, which gives the correspondence between the structure of a λc^λ-term and its type.

Lemma 4.1.16 (Generation lemma)

i) *If* $\Gamma;\Sigma \vdash u : \rho$ *then* $(u : \rho) \in \Gamma \cup \Sigma$.

ii) *If* $\Gamma;\Sigma \vdash \lambda x.U : \rho$ *then* $\rho = \mathtt{t}$ *and* $\Gamma, x : \mathtt{t};\Sigma \vdash U : \mathtt{t}$.

iii) *If* $\Gamma;\Sigma \vdash UV : \rho$ *then* $\rho = \mathtt{t}$, $\Gamma;\Sigma \vdash U : \mathtt{t}$ *and* $\Gamma;\Sigma \vdash V : \mathtt{t}$.

iv) *If* $\Gamma;\Sigma \vdash \Lambda\vec{x}.U : \rho$ *then*

either $\rho = [\vec{\mathtt{t}}]\mathtt{t}$ *and* $\Gamma, \vec{x} : \vec{\mathtt{t}};\Sigma \vdash U : \mathtt{t}$,

or $\rho = [\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t})$ *and* $\Gamma, \vec{x} : \vec{\mathtt{t}};\Sigma \vdash U : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$.

v) *If* $\Gamma;\Sigma \vdash U\langle\vec{U}\rangle : \rho$ *then* $\rho = \mathtt{t}$, $\Gamma;\Sigma \vdash U : [\vec{\mathtt{t}}]\mathtt{t}$ *and* $\Gamma;\Sigma \vdash \vec{U} : \vec{\mathtt{t}}$.

vi) *If* $\Gamma;\Sigma \vdash \delta\vec{h}.U : \rho$ *then* $\rho = [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$ *and* $\Gamma;\Sigma, h_1 : [\vec{\mathtt{t}}_1]\mathtt{t}, \ldots, h_n : [\vec{\mathtt{t}}_n]\,\mathtt{t} \vdash U : \mathtt{t}$.

vii) *If* $\Gamma;\Sigma \vdash U\lceil\vec{U}\rceil : \rho$ *then* $\rho = \mathtt{t}$, $\Gamma;\Sigma \vdash U : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$ *and* $\Gamma;\Sigma \vdash U_i : [\vec{\mathtt{t}}_i]\mathtt{t}$ *for* $1 \leq i \leq n$.

viii) If $\Gamma;\Sigma \vdash U \circ \vec{V} : \rho$ *then* $\rho = [\vec{\mathtt{t}}_{1,1}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{n,l_n}]\mathtt{t} \Rightarrow \mathtt{t}$, $\Gamma;\Sigma \vdash U : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$, *and* $\Gamma;\Sigma \vdash V_i : [\vec{\mathtt{t}}_i]([\vec{\mathtt{t}}_{i,1}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{i,l_i}]\mathtt{t} \Rightarrow \mathtt{t})$ *for* $1 \leq i \leq n$.

Proof: Suppose $\Gamma;\Sigma \vdash U : \rho$. The statements follow by distinguishing the cases of the structure of U. QED

Note though, that with the typing being a Curry-style typing (where abstractions are not annotated with types), not all typable λc-terms do have a unique type. For example, the λc-term $\delta h.\, x$ is typable by the types of the form $[\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}$ for any length of the vector $\vec{\mathtt{t}}$ (including 0). The type is not unique because the variable h does not occur in the term, so there is no indication of the arity of the hole which h represents. At the same time, this is the only cause for the loss of type uniqueness. (In the case of λ-abstractions and Λ-abstractions, the type of all the variables in the abstractions is implicitly $\mathtt{t}$.)

In the typing rules, the bases in the premisses are the same as the bases in the conclusions. Often it is the case that a smaller base is used for typing a subterm in the premisses. However, this fact does not restrain the typing derivations because both bases may be enlarged.

Lemma 4.1.17 (Bases thinning lemma)

i) If $\Gamma;\Sigma \vdash U : \rho$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma';\Sigma \vdash U : \rho$.

ii) If $\Gamma;\Sigma \vdash U : \rho$ *and* $\Sigma \subseteq \Sigma'$ *then* $\Gamma;\Sigma' \vdash U : \rho$.

Proof: Both proofs are conducted by induction to the length of $\Gamma;\Sigma \vdash U : \rho$. QED

The set of typable terms is closed under substitution. Even stronger, substitution preserves types, that is, if U and V are well-typed terms and if the substitution $[\![\vec{u} := \vec{V}]\!]$ respects types (i.e. for each i with $0 \leq i \leq |\vec{u}|$ if $u_i := V_i$ then u_i and V_i are of the same type), then $U[\![\vec{u} := \vec{V}]\!]$ is of the same type as U. In the next lemma, we use a comma instead of a semicolon as a separator between the bases, because $\vec{u} : \vec{\rho}$ may belong to the basis Γ as well as to the basis Σ.

Lemma 4.1.18 (Substitution lemma) *If* $\Gamma, \vec{u} : \vec{\rho}, \Sigma \vdash U : \rho$ *and* $\Gamma;\Sigma \vdash \vec{V} : \vec{\rho}$ *then* $\Gamma;\Sigma \vdash U[\![\vec{u} := \vec{V}]\!] : \rho$.

Proof: The proof is conducted by induction to U, using the Generation lemma. QED

The set of typable terms is closed under the rewrite relation in λc^{λ}. Like in the case of the substitution closure, even a stronger property holds: rewriting preserves types.

Lemma 4.1.19 (Subject reduction) *If* $\Gamma;\Sigma \vdash U : \rho$ *and* $U \twoheadrightarrow V$ *then* $\Gamma;\Sigma \vdash V : \rho$.

Proof: We sketch the proof of the statement only for $U \to V$; the statement follows then by induction to the length of the reduction $U \twoheadrightarrow V$.

Let $\Gamma; \Sigma \vdash U : \rho$ and $U \equiv C[L] \to_{\mathsf{r}} C[R] \equiv V$. One first shows that if $\Gamma'; \Sigma' \vdash L : \rho'$ then $\Gamma'; \Sigma' \vdash R : \rho'$. This is proved by case distinction on the rewrite rules and by using the Generation lemma, Bases thinning lemma and Substitution lemma. Then $\Gamma; \Sigma \vdash V : \rho$ follows by induction to C. QED

Theorem 4.1.20 (Subsystem λc^λ) *The underlying ARS $\mathcal{A}_{\lambda c^\lambda}$ of the calculus λc^λ is an indexed sub-ARS of the underlying ARS $\mathcal{A}_{\lambda c}$ of the context calculus λc.*

Proof: The ARS $\mathcal{A}_{\lambda c^\lambda}$ satisfies the conditions of the definition (Definition 1.1.14) of an indexed ARS of $\mathcal{A}_{\lambda c}$:

i) The set of terms of λc^λ is a subset of the terms of λc.

ii) Each rewrite relation of λc^λ is the restriction of the same rewrite relation in λc because the rewrite relations are generated by the same rewrite rule schemas.

iii) The set of terms of λc^λ is closed under each rewrite relation of λc, by the Subject reduction lemma 4.1.19.

QED

Properties of rewriting in λc^λ

We are interested in the confluence and normalisation properties of rewriting in λc^λ. The confluence property for λc^λ will be shown in a couple of sentences because it follows easily from the confluence property of the framework λc. The confluence property of the framework λc has been proved in Section 3.2.5 in detail. Contrary to the confluence property, investigating the normalisation of rewriting, and particularly showing the normalisation property of restricted, context-related rewriting in λc^λ will call for hard work.

Theorem 4.1.21 *The calculus λc^λ is confluent.*

Proof: Each pair of diverging rewrite sequences in λc^λ can be tiled in λc, because the rewriting in λc has the confluence property (Theorem 3.2.40). Because λc^λ is closed under rewriting by the Subject reduction lemma 4.2.8, the whole diagram is within λc^λ. QED

The calculus λc^λ is not strongly normalising. For example, the term $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$ is well-typed (by $\mathtt{t}$), and as we know, this term can be endlessly rewritten. In fact, the calculus λc^λ is not even weakly normalising. For example, the same term $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$ has no normal form at all. The absence of normalisation properties in λc^λ is not a surprise because the untyped lambda calculus is contained in λc^λ and it is well-known that the untyped lambda calculus is not weakly, and thus also not strongly, normalising.

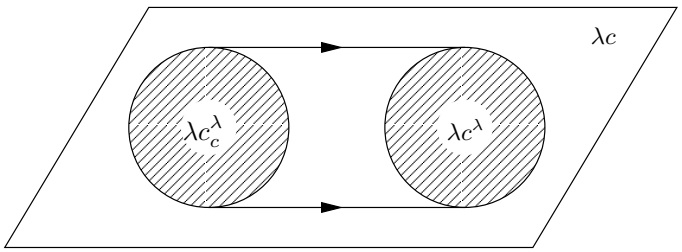


Figure 4.3: $\lambda c_c^\lambda \sqsubseteq \lambda c^\lambda \sqsubseteq \lambda c$

However, the rewriting related to contexts, that is, the rewriting generated by the rules (*fill*), ($\textcircled{m}\beta$) and ($\circ$), is strongly normalising. This is also not a surprise because the typing of contexts is in the style of simply typed lambda calculus, which is strongly normalising.

Before going any further, let us make precise what system we are talking about. Let the restriction of λc^λ to the rewriting related to contexts be called λc_c^λ.

Definition 4.1.22 (The calculus λc_c^λ) Let the calculus λc_c^λ be defined on the terms of λc^λ with the rewrite relations generated by the context rewrite rules: ($\textcircled{m}\beta$), (*fill*) and ($\circ$).

Notation. The rewriting of λc_c^λ will be denoted by $\to_c$ (cf. the remark after the definition of λc (Definition 3.1.5)).

Remark 4.1.23 The calculus λc_c^λ is a subsystem of the calculus λc^λ in the sense that the underlying ARS of λc_c^λ is an indexed sub-ARS of the underlying ARS $\mathcal{A}_{\lambda c^\lambda}$ of λc^λ. See Figure 4.3. One can easily check that the underlying ARS of λc_c^λ satisfies the definition of an indexed sub-ARS of $\mathcal{A}_{\lambda c^\lambda}$ (we keep in mind that the two calculi use the same rewrite rule schemas ($\textcircled{m}\beta$), (*fill*) and ($\circ$)):

i) both ARS have the same set of objects (i.e. $\textsc{Ter}(\lambda c^\lambda)$, which is generated from the set of λc-terms using the typing rules);

ii) the rewrite relations of λc_c^λ, which are generated by the rules ($\textcircled{m}\beta$), (*fill*) and ($\circ$), are the restrictions of the same rewrite relations of $\mathcal{A}_{\lambda c^\lambda}$; and

iii) the set of objects of λc_c^λ (i.e. $\textsc{Ter}(\lambda c^\lambda)$) is closed under rewriting of $\mathcal{A}_{\lambda c^\lambda}$, that is, it is closed under rewriting with respect to $\to_{\textcircled{m}\beta}$, $\to_{fill}$ and $\to_\circ$ (by Subject reduction lemma 4.1.19).

Back to the strong normalisation of λc_c^λ. Our proof of strong normalisation with respect to $\to_c$ is an adaptation of the standard proof of strong normalisation for the simply typed lambda calculus à la Church. The method of the standard proof is due to W. Tait (see [Tai67]).

The standard proof is conducted by induction to the term formation. Because the notion of strong normalisation is too weak for the induction, a strong-computability predicate on terms is employed to back the proof. Strong computability is defined by induction to types and it implies strong normalisation. The proof consists of two steps that show the following:

i) strong computability implies strong normalisation, and

ii) each well-typed term is strongly computable.

We use Tait's method to prove that $\rightarrow_c$ is strongly normalising on λc^{λ}-terms. For the most part we follow the presentation given by J.R. Hindley and J.P. Seldin (see Appendix 2 of [HS86]).

Before giving an overview of our proof, we look at some differences between $\lambda^{\rightarrow}$-Church and λc_c^{λ}.

- The calculus λc_c^{λ} deals with more terms than the calculus $\lambda^{\rightarrow}$-Church. Note that the set of λ-terms of $\lambda^{\rightarrow}$-Church is a subset of the set of λc_c^{λ}-terms, in the sense that each λ-term that can be typed in $\lambda^{\rightarrow}$-Church can also be typed in λc_c^{λ}. Moreover, the terms of λc_c^{λ} contain also the constructors Λ, $\langle\ \rangle$, δ, $\lceil\ \rceil$ and $\circ$. The constructors Λ and δ are variations of the constructor λ of lambda calculus; the constructors $\langle\ \rangle$ and $\lceil\ \rceil$ are variations of the application of lambda calculus.

- The calculus λc_c^{λ} deals with more rewrite relations than the calculus $\lambda^{\rightarrow}$-Church. The calculus contains the rewrite relations generated by the rules $(\textcircled{m}\beta)$ and (*fill*), which are variations of the rule β of lambda calculus, and thus, in particular, of $\lambda^{\rightarrow}$-Church. Furthermore, the calculus contains the rewrite relation generated by the rule $(\circ)$.

We adapt the standard proof in the following way. First, we will extend the definition of strong computability with respect to $\rightarrow_c$ to the terms of λc_c^{λ}. Then, we will adapt the proofs to the cases of new term constructors and of new rewrite rules. The adaptation regarding the constructor $\circ$ and the rule $(\circ)$ will be difficult. The adaptations regarding other constructors and rewrite rules will be trivial, because they all represent a variation of constructors and rewrite rules of lambda calculus.

Notation. If a term U is strongly normalising with respect to $\rightarrow_c$, we say that U is SN_c.

Definition 4.1.24 (Strong computability SC_c w.r.t. $\rightarrow_c$)

Strong computability with respect to $\rightarrow_c$ of λc^{λ}-terms is defined by induction to the type of terms as follows. If a term U is strongly computable w.r.t. $\rightarrow_c$, we say that U is SC_c.

i) Let $U : \mathtt{t}$. Then, U is SC_c if and only if U is strongly normalising with respect to $\rightarrow_c$.

ii) Let $U : [\vec{\mathtt{t}}]\mathtt{t}$. Then, U is SC_c if and only if for all strongly computable terms $\vec{V}$ with $V_i : \mathtt{t}$ for $1 \leq i \leq |\vec{\mathtt{t}}|$ the term $U\langle\vec{V}\rangle$ (of type $\mathtt{t}$) is SC_c.

iii) Let $U : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$. Then, U is SC_c if and only if for all strongly computable terms $\vec{V}$ with $V_i : [\vec{\mathtt{t}}_i]\mathtt{t}$ for $1 \leq i \leq n$ the term $U\lceil\vec{V}\rceil$ (of type $\mathtt{t}$) is SC_c.

iv) If $U : [\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t})$, then $U = \Lambda\vec{x}^{\vec{\mathtt{t}}}.U'$. In this case, U is SC_c if and only if for all SC_c terms $\vec{V}$ with $V_i : \mathtt{t}$ for $1 \leq i \leq |\vec{\mathtt{t}}|$ the term $U'[\![\vec{x} := \vec{V}]\!]$ (of type $[\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$) is SC_c.

Remark 4.1.25 The style of the strong computability definition is a mixture of two styles: the definition of strong computability presented in [HS86] on the one hand, and the definition of strong validity of Prawitz and the definition of computability of de Vrijer, on the other hand. In the former, a term of a functional type is SC_c if for all SC_c terms V, the term UV is SC_c. In the latter, a term of a functional type with $U \to \lambda x.U'$ is SC_c if for all SC_c terms V, the term $U'[\![x := V]\!]$ is SC_c. In the definition above, we follow the first style, except in the last clause. In the last clause we switch to the latter style, because, by following the former style we would obtain a term that is not typable: $(\Lambda\vec{x}^{\vec{\mathtt{t}}}.U')\langle\vec{V}\rangle$ where $U' : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$.

We start with proving the claim that variables are strongly computable with respect to $\to_c$. Indirectly, the same lemma claims that for each type ρ, there is a term of type ρ that is strongly computable with respect to $\to_c$.

Lemma 4.1.26 *Let u be a variable of type ρ. Then u is SC_c.*

Proof: In λc^{λ} there are two sorts of variables: term variables of type $\mathtt{t}$ and hole variables of type $[\vec{\mathtt{t}}]\mathtt{t}$.

If $u : \mathtt{t}$ then u is SN_c. Hence, u is SC_c by the definition of SC_c.

Otherwise, let $u : [\vec{\mathtt{t}}]\mathtt{t}$. In addition, let $\vec{W}$ be SC_c terms all of type $\mathtt{t}$. By the definition of SC_c, all $\vec{W}$ are SN_c. Then the term $u\langle\vec{W}\rangle$ is also SN_c. This term is of type $\mathtt{t}$, so it is SC_c. Then u is also SC_c, by definition. QED

Lemma 4.1.27 *Let U be a term of λc^{λ}. If U is SC_c then U is SN_c.*

Proof: The proof is conducted by induction to the type of U, using Lemma 4.1.26. QED

For the proof that each λc^{λ}-term is strongly computable a rather technical lemma is needed. This lemma states that if a reduct of a redex is strongly computable, then the redex is strongly computable too. In order to prove that, we need yet another lemma, saying that each λc^{λ}-term of type $[\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$ has a head normal form with respect to $\to_c$ of a special form.

Lemma 4.1.28 *Let $C : [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$. Then $C \twoheadrightarrow_c \delta\vec{h}.U$ with $h_i : [\vec{\mathtt{t}}_i]\mathtt{t}$ for $1 \leq i \leq |\vec{h}| = n$ and $U : \mathtt{t}$.*

Proof: By studying the typing rules, one can see that each term C of type $[\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$ has the form

$$C ::= \delta\vec{k}.V \mid C' \circ (\Lambda\vec{x}_1.C_1, \ldots, \Lambda\vec{x}_n.C_n)$$

where $V : \mathtt{t}$ and $C', C_1, \ldots, C_n$ have the same form as C. It can be shown that such a term C reduces by an innermost c-reduction to a term of the form $\delta\vec{h}.U$. The proof is conducted by induction to the number of symbols $\circ$ in C. The types of $\vec{h}$ and U follow by the Generation lemma. QED

Lemma 4.1.29

i) *Let $U, \vec{x}, \vec{W}$ be terms, all of type $\mathtt{t}$ and with $|\vec{x}| = |\vec{W}|$. Let W_i be SC_c if $x_i \notin FV(U)$, for $1 \leq i \leq |\vec{x}|$. If $U[\![\vec{x} := \vec{W}]\!]$ is SC_c then $(\Lambda\vec{x}.U)\langle\vec{W}\rangle$ is SC_c.*

ii) *Let $U : \mathtt{t}$, let $\vec{h}$ be n variables, and let $\vec{W}$ be n terms with h_i and W_i of the same type $[\vec{\mathtt{t}}_i]\mathtt{t}$, for $1 \leq i \leq n$. Let W_i be SC_c if $h_i \notin FV(U)$, for $1 \leq i \leq n$. If $U[\![\vec{h} := \vec{W}]\!]$ is SC_c then $(\delta\vec{h}.U)\lceil\vec{W}\rceil$ is SC_c.*

iii) *Let*

$$\begin{array}{lcll} C & : & [\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}, & \\ D_i & : & [\vec{\mathtt{t}}_{i,1}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{i,l_i}]\mathtt{t} \Rightarrow \mathtt{t}, & \\ x_i & : & \vec{\mathtt{t}}_i, & \\ \vec{M} & = & \vec{M}_1, \ldots, \vec{M}_n, & \\ \vec{M}_i & = & M_{i,1}, \ldots, M_{i,l_i}, & \\ M_{i,j} & : & [\vec{\mathtt{t}}_{i,j}]\mathtt{t}, & \textit{with } 1 \leq i \leq n, 1 \leq j \leq l_i. \end{array}$$

Let C, $(\Lambda\vec{x}_1.D_1)$, ..., $(\Lambda\vec{x}_n.D_n)$,$\vec{M}$ be SC_c. If the term $C\lceil\Lambda\vec{x}_1.D_1\lceil\vec{M}_1\rceil, \ldots, \Lambda\vec{x}_n.D_n\lceil\vec{M}_n\rceil\rceil$ is SC_c then the term $(C \circ (\Lambda\vec{x}_1.D_1, \ldots, \Lambda\vec{x}_n.D_n))\lceil\vec{M}\rceil$ is SC_c.

Proof:

i) Let $U[\![\vec{x} := \vec{W}]\!]$ be SC_c. By Lemma 4.1.27, this term is also SN_c. Hence, its subterms, U and all $\vec{W}$, are also SN_c. (If W_i does not occur in the term $U[\![\vec{x} := \vec{W}]\!]$, then $x_i \notin FV(U)$. In that case, use the extra assumption that W_i is SC_c and the same lemma.)

Suppose $(\Lambda\vec{x}.U)\langle\vec{W}\rangle$ has an infinite c-reduction. Then, since U and $\vec{W}$ are SN_c, eventually the head redex has to be reduced:

$$(\Lambda\vec{x}.U)\langle\vec{W}\rangle \twoheadrightarrow_c (\Lambda\vec{x}.U')\langle\vec{W}'\rangle \rightarrow_{\textcircled{m}\beta} U'[\![\vec{x} := \vec{W}']\!] \twoheadrightarrow_c \ldots (\infty)$$

Then $U[\![\vec{x} := \vec{W}]\!]$ has an infinite reduction too:

$$U[\![\vec{x} := \vec{W}]\!] \twoheadrightarrow_c U'[\![\vec{x} := \vec{W}']\!] \twoheadrightarrow_c \ldots (\infty)$$

This is a contradiction with the fact that $U[\![\vec{x} := \vec{W}]\!]$ is SN_c.

Then, $(\Lambda\vec{x}.U)\langle\vec{W}\rangle$ is SN_c. Because this term is of type $\mathtt{t}$, it is SC_c by the definition of SC_c.

ii) Analogously to the previous case.

iii) Let $C\lceil\Lambda\vec{x}_1.D_1\lceil\vec{M}_1\rceil,\ldots,\Lambda\vec{x}_n.D_n\lceil\vec{M}_n\rceil\rceil$ be SC_c. Then this term is SN_c, by Lemma 4.1.27.

Note that the previous two cases needed an additional assumption that W_i's are SC_c if they do not occur in the reduct $U[\![\vec{u} := \vec{W}]\!]$, because in this term W_i's occur only in the meta-level substitution. Such an assumption is not necessary in this case, because all D_i's and $M_{i,j}$'s occur as subterms in the term $C\lceil\Lambda\vec{x}_1.D_1\lceil\vec{M}_1\rceil,\ldots,\Lambda\vec{x}_n.D_n\lceil\vec{M}_n\rceil\rceil$.

Suppose the term $(C \circ (\Lambda\vec{x}_1.D_1,\ldots,\Lambda\vec{x}_n.D_n))\lceil\vec{M}_1,\ldots,\vec{M}_n\rceil$ has an infinite reduction. By Lemma 4.1.27 and the assumption that C, $(\Lambda\vec{x}_1.D_1)$, ..., $(\Lambda\vec{x}_n.D_n)$ and $\vec{M}$ are SC_c, these terms are also SN_c.

Moreover, by Lemma 4.1.28 the terms C and D_i's reduce to hole abstractions:

$$\begin{array}{lll} C & \twoheadrightarrow_c & \delta\vec{h}.N \\ D_i & \twoheadrightarrow_c & \delta\vec{g}_i.N_i \quad \text{for } 1 \le i \le n. \end{array}$$

Because the calculus λc^{λ} is confluent, these are the only head normal forms of these terms.

Then eventually, in this infinite reduction the head redex (with $\circ$ as the head symbol) has to be reduced:

$$\begin{array}{ll} & (C \circ (\Lambda\vec{x}_1.D_1,\ldots,\Lambda\vec{x}_n.D_n))\lceil\vec{M}_1,\ldots,\vec{M}_n\rceil \\ \twoheadrightarrow_c & ((\delta\vec{h}.N) \circ (\Lambda\vec{x}_1.\delta\vec{g}_1.N_1,\ldots,\Lambda\vec{x}_n.\delta\vec{g}_n.N_n))\lceil\vec{M}'_1,\ldots,\vec{M}'_n\rceil \\ \to_{\circ} & (\delta\vec{g}_1,\ldots,\vec{g}_n.N[\![h_1 := \Lambda\vec{x}_1.N_1,\ldots,h_n := \Lambda\vec{x}_n.N_n]\!])\lceil\vec{M}'_1,\ldots,\vec{M}'_n\rceil \end{array}$$

with $\vec{g} = \vec{g}_1,\ldots,\vec{g}_n \notin \mathrm{FVAR}(N)$. Without loss of generality, we assume that $\vec{g}_i \notin \mathrm{FVAR}(N_j)$ if $i \neq j$ for $1 \le i,j \le n$. At this stage of the proof we only state the following claim, and prove it later on.

> **Claim:** $N[\![h_1 := \Lambda\vec{x}_1.N_1,\ldots,h_n := \Lambda\vec{x}_n.N_n]\!]$ has no infinite reductions.

Then eventually, the head redex has to be reduced:

$$\begin{array}{ll} & (\delta\vec{g}_1,\ldots,\vec{g}_n.N[\![h_1 := \Lambda\vec{x}_1.N_1,\ldots,h_n := \Lambda\vec{x}_n.N_n]\!])\lceil\vec{M}'_1,\ldots,\vec{M}'_n\rceil \\ \twoheadrightarrow_c & (\delta\vec{g}_1,\ldots,\vec{g}_n.N')\lceil\vec{M}''_1,\ldots,\vec{M}''_n\rceil \\ \to_{\mathit{fill}} & N'[\![\vec{g}_1 := \vec{M}''_1,\ldots,\vec{g}_n := \vec{M}''_n]\!] \\ \twoheadrightarrow_c & \ldots\ (\infty) \end{array}$$

By summing up the rewrite sequences that took place in the infinite reduction, we also have (recall that $\vec{g} \notin \mathrm{FVAR}(N)$)

$$\begin{array}{ll} & N[\![h_1 := \Lambda\vec{x}_1.\, N_1[\![\vec{g} := \vec{M}]\!], \ldots h_n := \Lambda\vec{x}_n.\, N_n[\![\vec{g} := \vec{M}]\!]]\!] \\ = & N\,[\![h_1 := \Lambda\vec{x}_1.\, N_1, \ldots, h_n := \Lambda\vec{x}_n.\, N_n]\!]\,[\![\vec{g} := \vec{M}]\!] \\ \twoheadrightarrow_c & N'\,[\![\vec{g} := \vec{M}]\!]\,. \end{array}$$

But then we also have the following infinite reduction:

$$\begin{array}{ll} & C\lceil\Lambda\vec{x}_1.\, D_1\lceil\vec{M}_1\rceil, \ldots, \Lambda\vec{x}_n.\, D_n\lceil\vec{M}_n\rceil\rceil \\ \twoheadrightarrow_c & (\delta\vec{h}.\, N)\lceil\Lambda\vec{x}_1.\, (\delta\vec{g}_1.\, N_1)\lceil\vec{M}_1''\rceil, \ldots, \Lambda\vec{x}_n.\, (\delta\vec{g}_n.\, N_n)\lceil\vec{M}_n''\rceil\rceil \\ \twoheadrightarrow_{\mathit{fill}} & (\delta\vec{h}.\, N)\lceil\Lambda\vec{x}_1.\, N_1[\![\vec{g}_1 := \vec{M}_1'']\!], \ldots, \Lambda\vec{x}_n.\, N_n[\![\vec{g}_n := \vec{M}_n'']\!]\rceil \\ \to_{\mathit{fill}} & N[\![h_1 := \Lambda\vec{x}_1.\, N_1[\![\vec{g}_1 := \vec{M}_1'']\!], \ldots, h_n := \Lambda\vec{x}_n.\, N_n[\![\vec{g}_n := \vec{M}_n'']\!]]\!] \\ = & N\,[\![h_1 := \Lambda\vec{x}_1.\, N_1, \ldots, h_n := \Lambda\vec{x}_n.\, N_n]\!]\,[\![\vec{g} := \vec{M}'']\!] \\ \twoheadrightarrow_c & \ldots\ (\infty) \end{array}$$

This is a contradiction with the fact that the term $C\lceil\Lambda\vec{x}_1.\, D_1\lceil\vec{M}_1\rceil, \ldots, \Lambda\vec{x}_n.\, D_n\lceil\vec{M}_n\rceil\rceil$ is SN_c.

In conclusion, the term $(C \circ (\Lambda\vec{x}_1.\, D_1, \ldots, \Lambda\vec{x}_n.\, D_n))\lceil\vec{M}^1, \ldots, \vec{M}^n\rceil$ is SN_c. Because this term is of type $\mathtt{t}$, it is also SC_c by the definition of SC_c.

Proof of the claim: We recall the assumptions of the lemma and assumptions made at the stage of the proof where the claim is used:

$$\begin{array}{lll} C, (\Lambda\vec{x}_i.\, D_i) \text{ are all } SC_c, & & \\ C \quad \twoheadrightarrow_c \quad \delta\vec{h}.\, N & \text{where } C \text{ is } SN_c, & \\ D_i \quad \twoheadrightarrow_c \quad \delta\vec{g}_i.\, N_i & \text{where } N_i \text{ is } SN_c, \text{ for } 1 \leq i \leq n. & \end{array}$$

Moreover, check that $(1 \leq i \leq n)$:

$$\begin{array}{lll} C \text{ is } SC_c & \Leftrightarrow & \forall\vec{K} \text{ all } SC_c, \text{ we have } C\lceil\vec{K}\rceil \text{ is } SN_c. \quad (1) \\ \Lambda\vec{x}_i.\, D_i \text{ is } SC_c & \Rightarrow & \forall\vec{X}_i \text{ all } SC_c, \text{ we have } D_i\,[\![\vec{x}_i := \vec{X}_i]\!] \text{ is } SN_c. \quad (2) \end{array}$$

We prove that $\Lambda\vec{x}_i.\, N_i$ is SC_c for $1 \leq i \leq n$. So, let $\vec{X}_i$ be SC_c and suppose $(\Lambda\vec{x}_i.\, N_i)\langle\vec{X}_i\rangle$ has an infinite reduction. Because $\vec{X}_i$ are all SC_c and of type $\mathtt{t}$, the terms $\vec{X}_i$ are SN_c by the definition of SC_c. Because N_i and $\vec{X}_i$ are all SN_c, eventually the head redex is reduced:

$$(\Lambda\vec{x}_i.\, N_i)\langle\vec{X}_i\rangle \twoheadrightarrow_c\ (\Lambda\vec{x}_i.\, N_i')\langle\vec{X}_i'\rangle\ \to_{\textcircled{m}\beta}\ N_i'\,[\![\vec{x}_i := \vec{X}_i']\!]\ \twoheadrightarrow_c \ldots\ (\infty)$$

Then

$$D_i\,[\![\vec{x}_i := \vec{X}_i']\!]\ \twoheadrightarrow_c\ \delta\vec{g}_i.\, N_i'[\![\vec{x}_i := \vec{X}_i']\!]\ \twoheadrightarrow_c \ldots\ (\infty)$$

This is a contradiction with (2).

Then $(\Lambda\vec{x}_i.\, N_i)\langle\vec{X}_i\rangle$ must be SN_c. Then it is also SC_c, by the definition of SC_c. Then $\Lambda\vec{x}_i.\, N_i$ is SC_c, by the definition of SC_c.

By the fact (1) we know that $C\lceil\Lambda\vec{x}_1.\, N_1, \ldots, \Lambda\vec{x}_n.\, N_n\rceil$ cannot have an infinite reduction. One may check that

$$\begin{array}{ll} & C\lceil\Lambda\vec{x}_1.\, N_1, \ldots, \Lambda\vec{x}_n.\, N_n\rceil \\ \twoheadrightarrow_c & (\delta\vec{h}.\, N)\lceil\Lambda\vec{x}_1.\, N_1', \ldots, \Lambda\vec{x}_n.\, N_n'\rceil \\ \twoheadrightarrow_c & N\,[\![h_1 := \Lambda\vec{x}_1.\, N_1', \ldots, h_n := \Lambda\vec{x}_n.\, N_n']\!]\,. \end{array}$$

Then $N[\![h_1 := \Lambda\vec{x}_1.\, N_1', \ldots, h_n := \Lambda\vec{x}_n.\, N_n']\!]$ has no infinite reductions either.

QED

Lemma 4.1.30 *Let U be a typable term. Then, U is SC_c.*

Proof: The proof is conducted by induction to U. In fact, for the induction step to work, we need to prove the following strengthening of the lemma:

> Let $\vec{u}$ be $|\vec{V}|$ variables such that u_i and V_i are of the same type for $1 \leq i \leq |\vec{V}|$. If $\vec{V}$ are SC_c then $U[\![\vec{u} := \vec{V}]\!]$ is SC_c.

So let $\vec{u}$ and $\vec{V}$ be as required. Let $W^\star$ denote $W[\![\vec{u} := \vec{V}]\!]$. We treat four cases.

$U \equiv u$**:** with $u \not\equiv u_i$ for all $1 \leq i \leq |\vec{u}|$.

Then $U^\star = u$. By Lemma 4.1.26, u is SC_c.

$U \equiv W_1\, W_2$**:** Then $U^\star = W_1^\star W_2^\star$. By the induction hypothesis, the terms $W_1^\star$ and $W_2^\star$ are SC_c. By Lemma 4.1.27, the terms $W_1^\star$ and $W_2^\star$ are SN_c. Then $W_1^\star W_2^\star$ is also SN_c, because no c-redexes can be created by forming an application. Note that $W_1^\star W_2^\star : \mathtt{t}$. Then by the definition of SC_c, the term $W_1^\star W_2^\star$ is SC_c.

$U \equiv \delta h.\, W$**:** Then $U^\star = \delta\vec{h}.\, W^\star$. Let $\vec{W}$ be $|\vec{h}|$ SC_c terms with W_i of the same type as h_i for $1 \leq i \leq |\vec{h}|$. Then $W\,[\![\vec{u} := \vec{V}]\!]\,[\![\vec{h} := \vec{W}]\!]$ is SC_c, by the induction hypothesis. That is, $W^\star[\![\vec{h} := \vec{W}]\!]$ is SC_c. By Lemma 4.1.29(ii), the term $(\delta h.\, W^\star)\lceil\vec{W}\rceil$ is SC_c. By the definition of SC_c, the term $\delta h.\, W^\star$ is SC_c.

$U \equiv W \circ \vec{U}$**:** Then $U \equiv W \circ (\Lambda\vec{x}_1.\, W_1, \ldots, \Lambda\vec{x}_k.\, W_k)$. Moreover, this term is of type $[\vec{\mathtt{t}}_1]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}$ for some $n \in \mathbb{N}$ (intuitively, n is the sum of the number of holes in $\vec{W}$).

Then $U^\star = W^\star \circ (\Lambda\vec{x}_1.\, W_1^\star, \ldots, \Lambda\vec{x}_k.\, W_k^\star)$. By the induction hypothesis, the terms $W^\star$ and $\Lambda\vec{x}_i.\, W_i^\star$ for $1 \leq i \leq k$ are SC_c.

Let $\vec{Z}$ be n SC_c terms with $Z_l : [\vec{\mathtt{t}}_l]\mathtt{t}$ for $1 \leq l \leq n$ where these types are the same as in the type of U. Without loss of generality, choose $\vec{Z}$ such that $\{\vec{x}_1, \ldots, \vec{x}_k\} \cap FV(\vec{Z}) = \emptyset$. We prove that

$$W^\star \circ (\Lambda\vec{x}_1.\, W_1^\star, \ldots, \Lambda\vec{x}_k.\, W_k^\star)\lceil\vec{Z}\rceil \text{ is } SC_c.$$

In order to prove that, we first check that (split $\vec{Z}$ into $\vec{Y}_1, \ldots, \vec{Y}_k$)

for all $1 \leq i \leq k$, the term $\Lambda\vec{x}_i.\, W_i^\star \lceil \vec{Y}_i \rceil$ is SC_c.

Let $\vec{X}_i$ be SC_c terms with $X_{i,j}$ of the same type as $x_{i,j}$ for $1 \leq j \leq |\vec{x}_i|$. Then by the induction hypothesis, the term $W_i\, [\![\vec{u} := \vec{V}]\!]\, [\![\vec{x}_i := \vec{X}_i]\!]$ is SC_c. That is, the term $W_i^\star [\![\vec{x}_i := \vec{X}_i]\!]$ is SC_c. Then for SC_c terms $\vec{Y}_i$ the term $(W_i^\star [\![\vec{x}_i := \vec{X}_i]\!]) \lceil \vec{Y}_i \rceil$ is SC_c, by the definition of SC_c.

Note that $(W_i^\star [\![\vec{x}_i := \vec{X}_i]\!]) \lceil \vec{Y}_i \rceil = (W_i^\star \lceil \vec{Y}_i \rceil)[\![\vec{x}_i := \vec{X}_i]\!]$, by the choice of $\vec{Y}_i$. Then by Lemma 4.1.29(i), the term $(\Lambda\vec{x}_i.\, W_i^\star \lceil \vec{Y}_i \rceil) \langle \vec{X}_i \rangle$ is SC_c.

So we have that, for all SC_c terms $\vec{X}_i$, the term $(\Lambda\vec{x}_i.\, W_i^\star \lceil \vec{Y}_i \rceil) \langle \vec{X}_i \rangle$ is SC_c. By the definition of SC_c, the term $\Lambda\vec{x}_i.\, W_i^\star \lceil \vec{Y}_i \rceil$ is SC_c. Then the term $W^\star \lceil \Lambda\vec{x}_1.\, W_1^\star \lceil \vec{Y}_1 \rceil, \ldots, \Lambda\vec{x}_k.\, W_k^\star \lceil \vec{Y}_k \rceil \rceil$ is SC_c because $W^\star$ is SC_c by the induction hypothesis. Then the term $W^\star \circ (\Lambda\vec{x}_1.\, W_1^\star, \ldots, \Lambda\vec{x}_k.\, W_k^\star) \lceil \vec{Z} \rceil$ is SC_c by Lemma 4.1.29(iii). Note that $W^\star$, $\Lambda\vec{x}_1.\, W_1^\star$, …, $\Lambda\vec{x}_k.\, W_k^\star$ and $\vec{Z}$ are SC_c, as required by the applied lemma.

Finally, by the definition of SC_c, the term $W^\star \circ (\Lambda\vec{x}_1.\, W_1^\star, \ldots, \Lambda\vec{x}_k.\, W_k^\star)$ is SC_c. QED

Theorem 4.1.31 (Strong normalisation of $\rightarrow_c$) *The rewriting with respect to $\rightarrow_c$ in λc^{λ} is strongly normalising.*

Proof: This theorem is a corollary of Lemma 4.1.27 and Lemma 4.1.30. QED

Notation. The normal form of a λc^{λ}-term U with respect to the context rewrite rules will be denoted by $U{\downarrow_c}$.

Corollary 4.1.32 *The rewriting with respect to $\rightarrow_c$ in λc^{λ} is complete.*

Adequacy of context representation in λc^{λ}

In the remainder of the section we address the adequacy of the λ-context representation within λc^{λ}. Adequacy can be formulated as a couple of questions, which will be answered by the propositions that follow. The adequacy questions are:

i) Can λ-contexts and λ-terms be represented within λc^{λ}? Related to this question is the question whether every λc^{λ}-term corresponds to 'something' in lambda calculus?

ii) Does the context-related rewriting (i.e. $\rightarrow_c$) in λc^{λ} implement the context-related meta-operations (i.e. hole filling and composition) of lambda calculus?

iii) In the calculus λc^{λ}, does the context-related rewriting $\rightarrow_c$ disturb β-rewriting?

The basis of the correspondence between the untyped lambda calculus with contexts and the calculus λc^{λ} has been provided by the definitions of lambda objects (Definition 4.1.3) and translation of lambda objects into the framework λc (Definition 4.1.5). We will here show that this translation is actually a function from lambda objects into the calculus λc^{λ}. Using the translation function we will investigate some properties of representations of λ-terms and λ-contexts, and in this way answer the adequacy questions one by one.

Translations of (expressions resulting in) λ-terms and λ-contexts are indeed typable in λc^{λ} by specific types, as is stated in the next proposition. This assures that translation is well-defined and moreover affirms the intuition behind the types we gave in the subsection about the calculus λc^{λ}. At the same time, this proposition answers the first adequacy question about whether lambda objects can be represented within λc^{λ}.

Proposition 4.1.33 *Let R and P be expressions resulting in a λ-term and a λ-context, respectively. Then,*

i) $\Gamma \vdash {}^{\mathbb{\lceil}}R^{\mathbb{\rceil}} : \mathtt{t}$ *for a certain basis* Γ*; and*

ii) $\Gamma \vdash {}^{\mathbb{\lceil}}P^{\mathbb{\rceil}} : [\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}$ *for a certain basis* Γ.

Proof: One first proves that $\Gamma \vdash {}^{\mathbb{\lceil}}M^{\mathbb{\rceil}} : \mathtt{t}$ and $\Gamma \vdash {}^{\mathbb{\lceil}}C^{\mathbb{\rceil}} : [\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}$ for term M and context C as defined in Definition 4.1.1. Then, the proof of the statement is conducted by simultaneous induction on the structure of arbitrary lambda objects R and P. QED

From the point of view of λc^{λ}, the c-normal forms of type $\mathtt{t}$ which contain variables only of type $\mathtt{t}$ (i.e. variables only for representations of other λ-terms) are representations of the untyped λ-terms. This is proved in the next proposition.

Proposition 4.1.34 *If $\Gamma \vdash U : \mathtt{t}$, then there is M such that ${}^{\mathbb{\lceil}}M^{\mathbb{\rceil}} = U{\downarrow_c}$.*

Proof: One first shows that $U{\downarrow_c}$ does not contain hole variables or any of the symbols $\circ$, $\lceil$, δ, $\langle\rangle$, Λ, as follows. In $U{\downarrow_c}$ there are no free hole variables since $\Sigma = \emptyset$. Second, $U{\downarrow_c}$ contains none of the symbols $\circ$, $\lceil$ or δ, because in that case $U{\downarrow_c}$ would contain a *fill*-redex or a $\circ$-redex. Moreover, there are no bound hole variables because there are no hole binders δ in this term. Finally, $U{\downarrow_c}$ contains no $\langle\rangle$ or Λ: the subterms of the form $V\langle\vec{V}\rangle$ are $\textcircled{m}\beta$-redexes, since $V \not\equiv h$, and the subterms of the form $\Lambda\vec{x}.V$ occur only as a subterm of a $\textcircled{m}\beta$-redex. Then $U{\downarrow_c}$ is a λ-term, so $M = {}^{\mathbb{\lceil}}M^{\mathbb{\rceil}} = U{\downarrow_c}$. QED

A property analogous to Proposition 4.1.34 does not hold for the representations of λ-contexts, because in λc^{λ} one can represent β-reducts of (the translation of) a λ-context. For example, $U \equiv \delta h, k.\, h\langle k\langle x\rangle\rangle$ is a c-normal form of a 'context type' with free variables of 'term type', and for $C \equiv (\lambda y.\,[\,])(\lambda x.\,[\,])$ we have ${}^{\mathbb{\lceil}}C^{\mathbb{\rceil}} \twoheadrightarrow_{\beta} U$. However there is no λ-context D with ${}^{\mathbb{\lceil}}D^{\mathbb{\rceil}} = U$. Moreover, subterms like $h\langle k\langle x\rangle\rangle$ and $k\langle x\rangle$ have no meaning in the lambda calculus but they can be a subterm of

something meaningful. In sum, the calculus λc^λ contains λ-terms, λ-contexts and subresults of meta-computation, that is, it contains no 'junk'. This answers the question related to the first adequacy question.

The next proposition claims that the two ways of computing the translation of a complex lambda object $\llbracket P \rrbracket$ result in the same λc-term. That is, first evaluating the lambda object P to P^* and then translating P^* to λc^λ results in the same term as first translating P to λc^λ and then reducing the context-related redexes. This proposition assures that the context rewriting $\to_c$ in λc^λ correctly implements the meta-operations of hole filling and composition of lambda calculus. Thus, this proposition answers the second adequacy question.

Proposition 4.1.35 *Let P be a lambda object. Then $\llbracket P^* \rrbracket = \llbracket P \rrbracket{\downarrow_c}$.*

Proof: The proof is conducted by induction to P. QED

The last proposition states that in λc^λ hole filling and composition may freely be combined with β-reduction. This proposition answers the third adequacy question. Recall that in lambda calculus, β-steps may be performed in a λ-context only after filling the holes of the λ-context with λ-terms.

Proposition 4.1.36 *Let R be a lambda object that evaluates to a λ-term. Let $\llbracket R \rrbracket \twoheadrightarrow_\beta U$ and $\llbracket R \rrbracket \twoheadrightarrow_c V$. Then there is a λc^λ-term W such that $U \twoheadrightarrow_c W$ and $V \twoheadrightarrow_\beta W$.*

Proof: By Proposition 3.2.39, the rewrite relations $\to_c$ and $\to_\beta$ commute with each other. By Proposition 1.1.16, the rewriting in λc^λ inherits this property from λc because λc^λ is a subsystem of λc in the sense of Definition 1.1.14. QED

The introductory example in λc^λ

We close this section by showing how the problems with contexts in lambda calculus, which were illustrated in the introductory example 2.2.1, can be solved in λc^λ.

Let $C \equiv (\lambda y.\,[])\,x$ and $M \equiv x\,y$. Then the translation of the hole filling $C[M]$ is $\llbracket C[M] \rrbracket = (\delta g.\,(\lambda y.\,g\,\langle y \rangle)\,x)\,\lceil \Lambda y.\,x\,y \rceil$. This λc-term is well-typed in λc^λ (this may be checked directly using Proposition 4.1.33(i)):

$$x : \mathtt{t} \;\vdash\; (\delta g.\,(\lambda y.\,g\,\langle y \rangle)\,x)\,\lceil \Lambda y.\,x\,y \rceil : \mathtt{t}.$$

In λc^λ, the order of computing hole filling and performing the β-step in C is irrelevant:

$$\begin{array}{ccc}
(\delta g.\,(\lambda y.\,g\,\langle y \rangle)\,x)\,\lceil \Lambda y.\,x\,y \rceil & \to_\beta & (\delta g.\,g\,\langle x \rangle)\,\lceil \Lambda y.\,x\,y \rceil \\
\downarrow_{\mathit{fill}} & & \downarrow_{\mathit{fill}} \\
 & & \downarrow_{\textcircled{\scriptsize m}\beta} \\
(\lambda y.\,(\Lambda y.\,x\,y)\,\langle y \rangle)\,x & \twoheadrightarrow_{\beta,\textcircled{\scriptsize m}\beta} & x\,x.
\end{array}$$

Here, $\twoheadrightarrow_{\beta,\textcircled{m}\beta}$ stands for $\rightarrow_\beta;\rightarrow_{\textcircled{m}\beta}$ or $\rightarrow_{\textcircled{m}\beta};\rightarrow_\beta$. This diagram is the same as in Example 2.5.3.

4.2 The calculus $\lambda c^{\rightarrow}$

The calculus $\lambda c^{\rightarrow}$ given in this section describes the simply typed lambda calculus $\lambda^{\rightarrow}$ with contexts (*i*) with many holes, which may occur arbitrary number of times, (*ii*) where holes are filled sequentially, and (*iii*) including context variables and functions ranging over (representations of) λ-contexts. By sequential filling of holes we mean that if there are many holes in a context, the holes are filled one by one, as opposed to filling the holes all at once as in λc^{λ}. The representation of meta-contexts of $\lambda^{\rightarrow}$ within $\lambda c^{\rightarrow}$ follows the description given in the introduction. The typing rules of $\lambda c^{\rightarrow}$ for the most part follow the typing rules of the calculus of M. Hashimoto and A. Ohori (cf. [HO98]).

In this section, we will first define the calculus $\lambda c^{\rightarrow}$ and show that it is a subsystem of the framework λc (in the sense of Definition 1.1.14). Then, we will show that rewriting in $\lambda c^{\rightarrow}$ is complete. Next we will briefly address the adequacy of the context representation in $\lambda c^{\rightarrow}$. Finally, we will compare $\lambda c^{\rightarrow}$ to the calculus of M. Hashimoto and A. Ohori and to the calculus λc^{λ}, and comment on the introductory example within $\lambda c^{\rightarrow}$.

The calculus $\lambda c^{\rightarrow}$, a subsystem of λc

The typing of $\lambda c^{\rightarrow}$ controls the functionality of terms and contexts as the typing of the simply typed lambda calculus does. In addition, it also controls, loosely speaking, the well-formedness and the typing of the context-related machinery.

The set of the types of $\lambda c^{\rightarrow}$ is an extension of the set of the types of $\lambda^{\rightarrow}$.

Definition 4.2.1 (Types of $\lambda c^{\rightarrow}$) Let $\mathcal{V}^{\rightarrow}$ denote the set of base types with $\mathtt{a} \in \mathcal{V}^{\rightarrow}$. The τ-types ($\tau \in \mathcal{T}_{\rightarrow}$) and the ρ-types ($\rho \in \mathcal{P}_{\rightarrow}$) are defined as

$$\tau ::= \mathtt{a} \mid \tau \rightarrow \tau \mid [\vec{\tau}]\tau \Rightarrow \tau \quad \text{and} \quad \rho ::= \tau \mid [\vec{\tau}]\tau.$$

Here, $\rightarrow$ associates to the right, $\rightarrow$ binds stronger than $[\,]$ and $[\,]$ binds stronger than $\Rightarrow$.

Similarly to the case in λc^{λ}, the new type constructors $[\,]$ and $\Rightarrow$ are introduced for better correspondence with the the pairs of term constructors of λc (namely, $[\,]$ for the pair Λ and $\langle\rangle$, and $\Rightarrow$ for the pair δ and $\lceil\rceil$). The τ-types are used for typing representations of λ-terms and λ-contexts, and the ρ-types are also used for typing communicating objects and holes.

The typing uses two bases: the basis Γ, which is a set of declarations of the form $x : \tau$; and the basis Σ, which is a set of declarations of the form $h : [\vec{\tau}]\tau$. The bases are split for the same reason as in λc^{λ}: the basis Γ contains the true variables (i.e. the place-holders for representations of λ-terms and -context), and the basis

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma;\Sigma\ \vdash\ x:\tau}$$

$$(abs)\quad \frac{\Gamma,x:\tau;\Sigma\ \vdash\ U:\tau'}{\Gamma;\Sigma\ \vdash\ (\lambda x^{\tau}.U):\tau\rightarrow\tau'}$$

$$(app)\quad \frac{\Gamma;\Sigma\ \vdash\ U:\tau\rightarrow\tau' \qquad \Gamma;\Sigma\ \vdash\ V:\tau}{\Gamma;\Sigma\ \vdash\ UV:\tau'}$$

$$(hvar)\quad \frac{(h:[\vec{\tau}]\tau)\in\Sigma}{\Gamma;\Sigma\ \vdash\ h:[\vec{\tau}]\tau}$$

$$(mabs)\quad \frac{\Gamma,\vec{x}:\vec{\tau};\Sigma\ \vdash\ U:\tau}{\Gamma;\Sigma\ \vdash\ (\Lambda\vec{x}^{\vec{\tau}}.U):[\vec{\tau}]\tau}$$

$$(mapp)\quad \frac{\Gamma;\Sigma\ \vdash\ U:[\vec{\tau}]\tau \qquad \Gamma;\Sigma\ \vdash\ \vec{V}:\vec{\tau}}{\Gamma;\Sigma\ \vdash\ U\langle\vec{V}\rangle:\tau}$$

$$(habs)\quad \frac{\Gamma;\Sigma,h:[\vec{\tau}]\tau\ \vdash\ U:\tau'}{\Gamma;\Sigma\ \vdash\ (\delta h^{[\vec{\tau}]\tau}.U):[\vec{\tau}]\tau\Rightarrow\tau'}$$

$$(fill)\quad \frac{\Gamma;\Sigma\ \vdash\ U:[\vec{\tau}]\tau\Rightarrow\tau' \qquad \Gamma;\Sigma\ \vdash\ V:[\vec{\tau}]\tau}{\Gamma;\Sigma\ \vdash\ U\lceil V\rceil:\tau'}$$

Figure 4.4: Type system for $\lambda c^{\rightarrow}$

Σ contains the hole variables (i.e. the markers for the beginning (abstraction) and endings (holes) of a context).

In the remainder, the following notation will be used: $\tau,\sigma,\tau',\vec{\tau}\ldots\in\mathcal{T}_{\rightarrow}$, $\rho,\rho'\in\mathcal{P}_{\rightarrow}$, and $\vec{U}:\vec{\tau}$ denotes the pointwise typing $U_i:\tau_i$ for $1\leq i\leq|\vec{\tau}|$.

Definition 4.2.2 (Type system for $\lambda c^{\rightarrow}$) A term $U\in\textsc{Ter}(\lambda c)$ is typable by ρ from the bases Γ,Σ, if $\Gamma;\Sigma\ \vdash\ U:\rho$ can be derived using the typing rules displayed in Figure 4.4. The set of well-typed λc-terms will be denoted by $\textsc{Ter}(\lambda c^{\rightarrow})$.

We comment on the typing rules. The rules (*var*), (*abs*) and (*app*) are the familiar Church-style typing rules for $\lambda^{\rightarrow}$, now ranging also over (representations of) λ-contexts. That means that by these typing rules also context variables and functions ranging over contexts are typable. The rules (*hvar*), (*habs*) and (*fill*) are their

respective counterparts dealing with (introducing, abstracting and filling of) hole variables. The rules (*mabs*) and (*mapp*) are used for typing communication, where the components (variables in the abstraction and arguments in the application) are all of τ-types, meaning that in $\lambda c^{\rightarrow}$ representations of λ-terms and λ-contexts may be involved in communication.

Note that in the typing rules, only the 'unary' hole abstractor δ and the binary hole filler $\lceil\ \rceil$ are employed. This is because this typing captures sequential hole abstraction and filling. Sequential hole abstraction and filling means that if a context has many holes, the holes are filled one by one, as for example in

$$(\delta h_1.\, \delta h_2.\, U)\lceil V_1\rceil\lceil V_2\rceil \rightarrow_{\mathit{fill}} (\delta h_2.\, U[\![h_1 := V_1]\!])\lceil V_2\rceil \rightarrow_{\mathit{fill}} U\,[\![h_1 := V_1]\!]\,[\![h_2 := V_2]\!],$$

rather than all at once, as for example in

$$(\delta h_1, h_2.\, U)\lceil V_1, V_2\rceil \rightarrow_{\mathit{fill}} U\,[\![h_1 := V_1, h_2 := V_2]\!]\,.$$

Due to sequential hole abstraction and filling, the typing rules are closer to the typing in simply typed lambda calculus than the typing rules of λc^{λ}, which hopefully makes the rules more lucid.

Figure 4.5 is an example of typing in $\lambda c^{\rightarrow}$.

Definition 4.2.3 ($\lambda c^{\rightarrow}$) The terms of $\lambda c^{\rightarrow}$ are the well-typed terms of λc according to Definition 4.2.2. The rewrite rules are the rules (β), ($\textcircled{m}\beta$) and (*fill*) of λc, now restricted to $\lambda c^{\rightarrow}$-terms.

i) The lambda calculus rewrite rule is:

$$(\lambda x^{\tau}.\, U)V \rightarrow U[\![x := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$(\Lambda \vec{x}^{\vec{\tau}}.\, U)\langle \vec{V}\rangle \rightarrow U[\![\vec{x} := \vec{V}]\!] \qquad (\textcircled{m}\beta)$$
$$(\delta h^{[\vec{\tau}]\tau}.\, U)\lceil V\rceil \rightarrow U[\![h := V]\!]. \qquad (\mathit{fill})$$

Definition 4.2.4 Let the ARS $\mathcal{A}_{\lambda c^{\rightarrow}}$ be

$$\mathcal{A}_{\lambda c^{\rightarrow}} = \langle \textsc{Ter}(\lambda c^{\rightarrow}), \rightarrow_{\beta}, \rightarrow_{\textcircled{m}\beta}, \rightarrow_{\mathit{fill}}\rangle.$$

We call the ARS $\mathcal{A}_{\lambda c^{\rightarrow}}$ the underlying ARS of the calculus $\lambda c^{\rightarrow}$.

The rewrite rules of $\lambda c^{\rightarrow}$ form only a subset of the rewrite rules of the framework λc. The rewrite rules of $\lambda c^{\rightarrow}$ include only one instance of the rewrite rule (*fill*) of λc, namely the one with index 1, because the typing rules exclude typing of δ and $\lceil\ \rceil$ for any other index.

Moreover, there is no composition in $\lambda c^{\rightarrow}$. This is because composition is definable: for every (context) U of type $[\vec{\tau}]\tau \Rightarrow \tau'$ and every (communicating context) V

$$\dfrac{\dfrac{\dfrac{(h:[\mathtt{a}]\mathtt{a})\in\{h:[\mathtt{a}]\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h:[\mathtt{a}]\mathtt{a}}\quad\dfrac{(x:\mathtt{a})\in\{z:\mathtt{a},x:\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash x:\mathtt{a}}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h\langle x\rangle:\mathtt{a}}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash\lambda x^{\mathtt{a}}.h\langle x\rangle:\mathtt{a}\to\mathtt{a}}\quad\begin{array}{c}\cdot\\\cdot\\\cdot\end{array}\quad\dfrac{\dfrac{(h:[\mathtt{a}]\mathtt{a})\in\{h:[\mathtt{a}]\mathtt{a}\}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h:[\mathtt{a}]\mathtt{a}}\quad\dfrac{(z:\mathtt{a})\in\{z:\mathtt{a}\}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash z:\mathtt{a}}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h\langle z\rangle:\mathtt{a}}}{\dfrac{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash(\lambda x^{\mathtt{a}}.h\langle x\rangle)(h\langle z\rangle):\mathtt{a}}{z:\mathtt{a}\vdash\lambda h^{[\mathtt{a}]\mathtt{a}}.(\lambda x^{\mathtt{a}}.h\langle x\rangle)(h\langle z\rangle):[\mathtt{a}]\mathtt{a}\Rightarrow\mathtt{a}}}$$

Figure 4.5: An example of typing in $\lambda c^{\rightarrow}$

of type $[\vec{\tau}]([\vec{\sigma}]\sigma\Rightarrow\tau)$ the following closed typable λc-term can act as a composition constructor in $\underline{comp}\ U\ V$,

$$\begin{array}{rcl}\underline{comp} & \equiv & \lambda c^{[\vec{\tau}]\tau\Rightarrow\tau'}.\delta d^{[\vec{\tau}]([\vec{\sigma}]\sigma\Rightarrow\tau)}.\delta g^{[\vec{\sigma}]\sigma}.c\lceil\Lambda\vec{x}^{\vec{\tau}}.(d\langle\vec{x}\rangle)\lceil g\rceil\rceil\\ & : & ([\vec{\tau}]\tau\Rightarrow\tau')\to([\vec{\tau}]([\vec{\sigma}]\sigma\Rightarrow\tau)\Rightarrow([\vec{\sigma}]\sigma\Rightarrow\tau')).\end{array}$$

Consequently, the composition constructor, typing rule and rewrite rule are omitted.

The calculus $\lambda c^{\rightarrow}$ is a subsystem of the framework λc

In order to prove that the calculus $\lambda c^{\rightarrow}$ is a subsystem of the framework λc, we show that $\lambda c^{\rightarrow}$ is closed under substitution and rewriting. Note that the closure under rewriting pertains only to the closure under the rewriting of the calculus $\lambda c^{\rightarrow}$, which is a restriction of the rewriting of the framework λc (see Definition 1.1.14). The proofs are the standard ones, as in the case of $\lambda^{\rightarrow}$ *à la* Church.

As a bonus, we first prove that each $\lambda c^{\rightarrow}$-term has a unique type.

Lemma 4.2.5 (Uniqueness of types) *If* $\Gamma;\Sigma\vdash U:\rho_1$ *and* $\Gamma;\Sigma\vdash U:\rho_2$ *then* $\rho_1=\rho_2$.

Proof: By induction on the length of the derivation. QED

Lemma 4.2.6 (Generation lemma)

i) *If* $\Gamma;\Sigma\vdash u:\rho$ *then* $(u:\rho)\in\Gamma\cup\Sigma$.

ii) *If* $\Gamma;\Sigma\vdash\lambda x^{\rho'}.U:\rho$ *then* $\rho'\in\mathcal{T}_{\rightarrow}$ *and there is a* $\tau\in\mathcal{T}_{\rightarrow}$ *such that* $\rho=\rho'\to\tau$ *and* $\Gamma,x:\rho';\Sigma\vdash U:\tau$.

iii) *If* $\Gamma;\Sigma\vdash U_1U_2:\rho$ *then there is a* $\tau\in\mathcal{T}_{\rightarrow}$ *such that* $\Gamma;\Sigma\vdash U_1:\tau\to\rho$ *and* $\Gamma;\Sigma\vdash U_2:\tau$.

iv) *If* $\Gamma;\Sigma\vdash\Lambda\vec{x}^{\vec{\rho}}.U:\rho$ *then* $\vec{\rho}\in\mathcal{T}_{\rightarrow}$, *there is a* $\tau\in\mathcal{T}_{\rightarrow}$ *such that* $\rho=[\vec{\rho}]\tau$ *and* $\Gamma,\vec{x}:\vec{\rho};\Sigma\vdash U:\tau$.

v) *If* $\Gamma;\Sigma\vdash U\langle\vec{U}\rangle:\rho$ *then* $\rho\in\mathcal{T}_{\rightarrow}$ *and there are* $\vec{\tau}\in\mathcal{T}_{\rightarrow}$ *such that* $\Gamma;\Sigma\vdash U:[\vec{\tau}]\rho$ *and* $\Gamma;\Sigma\vdash\vec{U}:\vec{\tau}$.

vi) *If* $\Gamma;\Sigma \vdash \delta h^{\rho'}.U : \rho$ *then* $\rho' = [\vec{\tau}]\tau$ *and there is a* $\tau' \in \mathcal{T}_{\rightarrow}$ *such that* $\rho = [\vec{\tau}]\tau \Rightarrow \tau'$ *and* $\Gamma;\Sigma, h : [\vec{\tau}]\tau \vdash U : \tau'$.

vii) *If* $\Gamma;\Sigma \vdash U_1\lceil U_2\rceil : \rho$ *then there are* $\vec{\tau}, \tau \in \mathcal{T}_{\rightarrow}$ *such that* $\Gamma;\Sigma \vdash U_1 : [\vec{\tau}]\tau \Rightarrow \rho$ *and* $\Gamma;\Sigma \vdash U_2 : [\vec{\tau}]\tau$.

Proof: Suppose $\Gamma;\Sigma \vdash U : \rho$. The statements follow by distinguishing the cases of the structure of U. QED

Lemma 4.2.7 (Substitution lemma) *If* $\Gamma, \vec{u} : \vec{\rho}, \Sigma \vdash U : \rho$ *and* $\Gamma;\Sigma \vdash \vec{V} : \vec{\rho}$ *then* $\Gamma;\Sigma \vdash U[\![\vec{u} := \vec{V}]\!] : \rho$

Proof: The proof is conducted by induction to U, using the Generation lemma. QED

Lemma 4.2.8 (Subject reduction) *If* $\Gamma;\Sigma \vdash U : \rho$ *and* $U \twoheadrightarrow V$, *then* $\Gamma;\Sigma \vdash V : \rho$.

Proof: The heart of the argument is that in each contraction $L \rightarrow R$ the left-hand side and the right-hand side have the same type, which is proved by using the Generation lemma and the Substitution lemma. QED

In sum, because the calculus $\lambda c^{\rightarrow}$, which is defined on a subset of λc-terms and a subset of rewrite relations of λc, is closed under rewriting. Hence, it is a subsystem of λc.

Theorem 4.2.9 (Subsystem $\lambda c^{\rightarrow}$) *The underlying ARS* $\mathcal{A}_{\lambda c^{\rightarrow}}$ *of the calculus* $\lambda c^{\rightarrow}$ *is an indexed sub-ARS of the underlying ARS* $\mathcal{A}_{\lambda c}$ *of the context calculus* λc.

Proof: One can easily check that the ARS $\mathcal{A}_{\lambda c^{\rightarrow}}$ satisfies the definition (Definition 1.1.14) of an indexed sub-ARS of $\mathcal{A}_{\lambda c}$. QED

Properties of rewriting in $\lambda c^{\rightarrow}$

Rewriting in $\lambda c^{\rightarrow}$ is confluent. The confluence property of $\lambda c^{\rightarrow}$ follows from a stronger property of λc than confluence, namely the commutation property of each pair of rewrite relations. The reason that a stronger property is needed lies in the fact that the rewrite relation of $\lambda c^{\rightarrow}$ is defined on a *subset* of the rewrite rules of λc. What is needed in the proof of confluence is that in the pair of converging rewrite sequences the same rewrite rules are used that occur in the pair of the diverging rewrite sequences, which are the rewrite rules of $\lambda c^{\rightarrow}$. This means that no other rewrite rules are needed to finish the confluence diagram than the rewrite rules of $\lambda c^{\rightarrow}$ (see Remark 3.2.2).

Theorem 4.2.10 *The calculus* $\lambda c^{\rightarrow}$ *is confluent.*

Proof: Each pair of diverging rewrite sequences in $\lambda c^{\to}$ can in λc be tiled by the commutation tiles, because each pair of (unions of) rewrite relations in λc commutes. In such a tiled diagram, the pair of converging rewrite sequences uses the same rewrite rules as the pair of diverging rewrite sequences. That means that the whole diagram is within $\lambda c^{\to}$. QED

Note that actually, the calculus $\lambda c^{\to}$ has the commutation property of pairs of rewrite relations too.

Rewriting in $\lambda c^{\to}$ is strongly normalising. The proof of strong normalisation can be done via the natural translation of $\lambda c^{\to}$ into $\lambda^{\to}$. Then the strong normalisation of rewriting in $\lambda c^{\to}$ follows from the strong normalisation of rewriting in $\lambda^{\to}$.

Here, $\textsc{Typ}(\lambda^{\to})$ denotes the types of the simply typed lambda calculus.

Definition 4.2.11 (Translation of $\lambda c^{\to}$ to $\lambda^{\to}$)

i) Define $\llbracket\rceil : \mathcal{P}_{\to} \to \textsc{Typ}(\lambda^{\to})$ as a function that translates the types to simple types:

$$\begin{array}{lcl}
\llbracket\, \mathtt{a} \,\rceil & = & \mathtt{a} \\
\llbracket \tau \to \tau' \rceil & = & \llbracket \tau \rceil \to \llbracket \tau' \rceil \\
\llbracket [\vec{\tau}]\tau \rceil & = & \llbracket \tau_1 \rceil \to \ldots \to \llbracket \tau_n \rceil \to \llbracket \tau \rceil \\
\llbracket [\vec{\tau}]\tau \Rightarrow \tau' \rceil & = & (\llbracket \tau_1 \rceil \to \ldots \to \llbracket \tau_n \rceil \to \llbracket \tau \rceil) \to \tau'.
\end{array}$$

ii) Define $\llbracket\rceil : \textsc{Ter}(\lambda c^{\to}) \to \textsc{Ter}(\lambda^{\to})$ as

$$\begin{array}{lcl}
\llbracket u \rceil & = & u \\
\llbracket\, \lambda x^{\tau}.U \,\rceil & = & \lambda x^{\llbracket \tau \rceil}.\llbracket U \rceil \\
\llbracket\, U_1 U_2 \,\rceil & = & \llbracket U_1 \rceil \llbracket U_2 \rceil \\
\llbracket\, \Lambda \vec{x}^{\vec{\tau}}.U \,\rceil & = & \lambda \vec{x}^{\llbracket \vec{\tau} \rceil}.\llbracket U \rceil \\
\llbracket\, U\langle \vec{U} \rangle \,\rceil & = & \llbracket U \rceil \llbracket U_1 \rceil \ldots \llbracket U_n \rceil \\
\llbracket\, \delta h^{[\vec{\tau}]\tau}.U \,\rceil & = & \lambda h^{\llbracket [\vec{\tau}]\tau \rceil}.\llbracket U \rceil \\
\llbracket\, U_1 \lceil U_2 \rceil \,\rceil & = & \llbracket U_1 \rceil \llbracket U_2 \rceil.
\end{array}$$

iii) Let Δ be a basis. Then $\llbracket \Delta \rceil = \{(u : \llbracket \rho \rceil) \mid (u : \rho) \in \Delta\}$.

Translation preserves typing.

Lemma 4.2.12 *If* $\Gamma; \Sigma \vdash_{\lambda c^{\to}} U : \rho$ *then* $\llbracket \Gamma \rceil \cup \llbracket \Sigma \rceil \vdash_{\lambda^{\to}} \llbracket U \rceil : \llbracket \rho \rceil$.

Proof: By induction to the length of $\Gamma; \Sigma \vdash_{\lambda c^{\to}} U : \rho$. Check the typing rules of λc: from the translations of the premisses, the translations of the conclusions can be derived in $\lambda^{\to}$. Then each derivation step in $\Gamma; \Sigma \vdash_{\lambda c^{\to}} U : \rho$ can be translated to one or more derivation steps in $\lambda^{\to}$. QED

Translation preserves rewrite steps.

Proposition 4.2.13 *If* $\Gamma; \Sigma \vdash_{\lambda c^{\to}} U : \rho$ *and* $U \to V$ *in* $\lambda c^{\to}$, *then* $\llbracket U \rceil \twoheadrightarrow \llbracket V \rceil$ *in the simply typed lambda calculus.*

Proof: In fact, the $\lambda c^{\rightarrow}$-rewrite steps are translated into a positive number of β-steps, with one exception: a $\textcircled{m}\beta$-step where the multiple abstraction and the multiple application are empty, that is, $(\Lambda\epsilon.U)\langle\rangle \rightarrow_{\textcircled{m}\beta} U$, which in translation results in an empty β-reduction: $\llbracket (\Lambda\epsilon.U)\langle\rangle \rrbracket = U = \llbracket U \rrbracket$.

The proof proceeds as follows. By Lemma 4.2.12, we know that $\llbracket U \rrbracket$ is a $\lambda^{\rightarrow}$-term. If $U \rightarrow V$ is an 'empty' $\textcircled{m}\beta$-step, then we are done. Otherwise, one shows that, in general, a redex (other than an 'empty' $\textcircled{m}\beta$-redex) of $\lambda c^{\rightarrow}$ translates into a redex of $\lambda^{\rightarrow}$. Because the set of $\lambda^{\rightarrow}$-terms is closed under rewriting, we know also that all reducts of $\llbracket U \rrbracket$, including $\llbracket V \rrbracket$, are $\lambda^{\rightarrow}$-terms. QED

Theorem 4.2.14 (Strong normalisation) *Rewriting in $\lambda c^{\rightarrow}$ is strongly normalising.*

Proof: Let $U_0 \in \textsc{Ter}(\lambda c^{\rightarrow})$ and suppose r is an infinite rewrite sequence in $\lambda c^{\rightarrow}$:

$$r: \quad U_0 \rightarrow U_1 \rightarrow U_2 \rightarrow \ldots \ (\infty).$$

Note indeed that if U_0 is a $\lambda c^{\rightarrow}$-term, then so are all its reducts. Then, the translation of U_i's to the simply typed lambda calculus results in a rewrite sequence $\llbracket r \rrbracket$ in the simply typed lambda calculus:

$$\llbracket r \rrbracket: \quad \llbracket U_0 \rrbracket \twoheadrightarrow \llbracket U_1 \rrbracket \twoheadrightarrow \llbracket U_2 \rrbracket \twoheadrightarrow \ldots \ (\infty).$$

Because there are no infinite rewrite sequences in $\lambda^{\rightarrow}$, the tail of $\llbracket r \rrbracket$ must eventually be empty, i.e. $\llbracket U_n \rrbracket \equiv \llbracket U_{n+1} \rrbracket \equiv \ldots$. These steps can only be translations of 'empty' $\textcircled{m}\beta$-steps, i.e. $U_n \equiv C[(\Lambda\epsilon.U)\langle\rangle] \rightarrow_{\textcircled{m}\beta} C[U] \equiv U_{n+1} \ldots$. However, since $\lambda c^{\rightarrow}$-terms are finite, there cannot be infinitely many such steps starting from U_n. QED

Corollary 4.2.15 *Rewriting in $\lambda c^{\rightarrow}$ is complete.*

Adequacy of context representation in $\lambda c^{\rightarrow}$

From the lambda calculus viewpoint, the $\lambda c^{\rightarrow}$-terms of a τ-type with free variables only of type τ are representations of λ-terms, λ-contexts, functions ranging over these elements. The other $\lambda c^{\rightarrow}$-terms are intermediate representations of λ-contexts and communicating objects. A more precise correspondence between the simply typed lambda calculus with contexts and the calculus $\lambda c^{\rightarrow}$ is difficult to pinpoint because in $\lambda c^{\rightarrow}$ we deal with notions and notations that are not defined or are inadequate in lambda calculus.

Example 4.2.16 Let $U \equiv \lambda c^{[\mathtt{a}]\mathtt{b} \Rightarrow \mathtt{a} \rightarrow \mathtt{b}}.\, \lambda y^{\mathtt{b}}.\, c\lceil \Lambda z^{\mathtt{a}}.\, y \rceil$. The term U can be seen as a function ranging over (a representation of) a context. In the calculus $\lambda c^{\rightarrow}$, the term $U\,(\delta h^{[\mathtt{a}]\mathtt{b}}.\, \lambda x^{\mathtt{a}}.\, h\langle x \rangle)$ reduces in two steps to $V \equiv \lambda y^{\mathtt{b}}.\, \lambda x^{\mathtt{a}}.\, (\Lambda z^{\mathtt{a}}.\, y)\langle x \rangle$. Neither the term U nor the term V can be denoted in the simply typed lambda calculus, due to the notational shortcomings mentioned above.

Comparison to other context calculi

The calculus described in this section extends the work of M. Hashimoto and A. Ohori (see [HO98], see also Section 2.4 about context calculi) in the following sense. It includes multiple occurrences of a hole and drops their condition on the β-rule, by which β-reduction is not allowed within (representations of) λ-contexts. Moreover, $\lambda c^{\rightarrow}$ allows composition, which is not present in their system. However, whereas the order of arguments $\vec{U}$ at the holes $h\langle\vec{U}\rangle$ is relevant in our calculus, in the calculus of M. Hashimoto and A. Ohori the order is irrelevant.

The calculus $\lambda c^{\rightarrow}$ can also be compared to the calculus λc^{λ}. There are two main differences between these calculi. The first difference lies in the fact that $\lambda c^{\rightarrow}$ is a calculus over contexts, while λc^{λ} offers only a method of context representation. For example, $\vdash_{\lambda c^{\rightarrow}} (\lambda c^{[\mathtt{a}]\mathtt{a}\Rightarrow\mathtt{b}}.\, c\lceil\Lambda x^{\mathtt{a}}.\, x\rceil) \;:\; (([\mathtt{a}]\mathtt{a}\Rightarrow\mathtt{b})\rightarrow\mathtt{b})$ where $\lambda c^{[\mathtt{a}]\mathtt{a}\Rightarrow\mathtt{b}}.\, c\lceil\Lambda x^{\mathtt{a}}.\, x\rceil$ represents a function ranging over a context, but this term is not typable in λc^{λ} (for any type decoration of variables in abstractions). The second difference is that $\lambda c^{\rightarrow}$ deals with the simply typed lambda calculus, whereas λc^{λ} deals with the untyped lambda calculus. For example, $\vdash_{\lambda c^{\lambda}} (\delta h.\, \lambda x.\, xx) \;:\; ([\mathtt{t}]\mathtt{t}\Rightarrow\mathtt{t})$, whereas the λc-term $\delta h.\, \lambda x.\, xx$ is not typable in $\lambda c^{\rightarrow}$ because the self-application xx is not typable. An additional minor difference is that the holes of a context are filled sequentially in $\lambda c^{\rightarrow}$, whereas they are filled simultaneously in λc^{λ}.

The introductory example in $\lambda c^{\rightarrow}$

The introductory example 2.2.1 cannot be represented within $\lambda c^{\rightarrow}$ because the example involves λ-terms that are not typable in $\lambda^{\rightarrow}$, and consequently also not typable in $\lambda c^{\rightarrow}$. Recall that $C \equiv (\lambda y.\, [])x$ and $M \equiv xy$. Then, in particular, $C[M] \rightarrow_{\beta} xx$ and the term xx is not typable in $\lambda^{\rightarrow}$, thus it is also not typable in $\lambda c^{\rightarrow}$.

4.3 The calculus $\lambda c^{\cong}$

The calculus $\lambda c^{\cong}$ defined in this section is a context calculus for untyped lambda calculus. This calculus supports λ-contexts with many holes, which may occur an arbitrary number of times, and which are filled sequentially. It also supports functions ranging over λ-contexts. Technically, it is a variation of the calculus $\lambda c^{\rightarrow}$ obtained by restricting the base types to only one constant $\mathtt{t}$ and imposing the equality $\mathtt{t}\rightarrow\mathtt{t}\cong\mathtt{t}$ on types. Such typing essentially has the effect of well-formedness rules on the untyped λ-terms and of typing rules on the contexts and holes. As such, this calculus describes a minimal typing necessary to ensure that the number and kind of arguments in the context machinery match. At the same time it places no constraints upon the formation of λ-terms (see also Remark 3.1.10). For this reasons, this kind of typing may be called syntactic, as in the case of λc^{λ}.

This section is structured as follows. We will first define the calculus $\lambda c^{\cong}$ by defining the type system and rewrite rules, and we will show that this calculus is a subsystem of the framework λc. Then we will show that $\lambda c^{\cong}$ has the confluence

property. The uniqueness of typing and strong normalisation of $\rightarrow$ are lost, as expected, due to the imposed equality on types and the presence of untyped lambda calculus, respectively. However, we will show that the rewriting which deals with hole filling, communication and functions ranging over contexts including composition is strongly normalising. Next, we will compare the calculus $\lambda c^{\cong}$ with λc^{λ} and $\lambda c^{\rightarrow}$. We will show that both λc^{λ} and $\lambda c^{\rightarrow}$ can be embedded into $\lambda c^{\cong}$ in such a way that the rewrite steps are preserved, but that $\lambda c^{\cong}$ is more expressive than λc^{λ} or $\lambda c^{\rightarrow}$. Finally, we comment on the introductory example within $\lambda c^{\cong}$.

Remark 4.3.1 The equality $\mathtt{t} \rightarrow \mathtt{t} \cong \mathtt{t}$ on types is *not* related to recursive types as for example in $\lambda\mu$ (cf. for example [Bar92]). This equality is a maneuver which is incorporated in order to ignore the types on (the representations of) λ-terms.

The calculus $\lambda c^{\cong}$

The types of $\lambda c^{\cong}$ are the same as in $\lambda c^{\rightarrow}$, only now over the singleton $\{\mathtt{t}\}$. These types will be considered modulo equality $\mathtt{t} \rightarrow \mathtt{t} \cong \mathtt{t}$. Such an approach to types will have the effect of ignoring the type constructor $\rightarrow$ on (representations of) λ-terms. Consequently, the term-abstraction and the term-application typing rules will impose the well-formedness on (representations of) λ-terms rather than the well-typedness as in the case of $\lambda c^{\rightarrow}$.

We now give the definition of types, the definition of congruence $\cong$ on types generated by $\mathtt{t} \rightarrow \mathtt{t} \cong \mathtt{t}$, and the definition of minimal types.

Definition 4.3.2 (Types of $\lambda c^{\cong}$) Let the set of base types be the singleton $\mathcal{V}^{\cong} = \{\mathtt{t}\}$. The τ-types ($\tau \in \mathcal{T}_{\cong}$) and the ρ-types ($\rho \in \mathcal{P}_{\cong}$) are defined as

$$\tau ::= \mathtt{t} \mid \tau \rightarrow \tau \mid [\vec{\tau}]\tau \Rightarrow \tau \quad \text{and} \quad \rho ::= \tau \mid [\vec{\tau}]\tau.$$

Here, $\rightarrow$ associates to the right, $\rightarrow$ binds stronger than $[\,]$ and $[\,]$ binds stronger than $\Rightarrow$.

The congruence relation on types is generated by the equality $\mathtt{t} \rightarrow \mathtt{t} \cong \mathtt{t}$.

Definition 4.3.3 (Congruence) Let $\rho, \rho' \in \mathcal{P}_{\cong}$ and let $\tau, \sigma, \vec{\tau}, \vec{\tau}', \tau', \sigma' \in \mathcal{T}_{\cong}$. The congruence relation $\cong$ on types is defined by the following axiom and rules.

$$(axiom) \quad \mathtt{t} \rightarrow \mathtt{t} \cong \mathtt{t}$$

$$(cong) \quad \frac{\sigma \cong \sigma' \quad \vec{\tau} \cong \vec{\tau}' \quad \tau \cong \tau'}{\begin{array}{rcl} \sigma \rightarrow \tau & \cong & \sigma' \rightarrow \tau' \\ [\vec{\tau}]\sigma & \cong & [\vec{\tau}']\sigma' \\ [\vec{\tau}]\tau \Rightarrow \sigma & \cong & [\vec{\tau}']\tau' \Rightarrow \sigma' \end{array}}$$

$$(refl) \quad \rho \cong \rho$$

$$(sym) \quad \frac{\rho \cong \rho'}{\rho' \cong \rho}$$

$$(trans) \quad \frac{\rho_1 \cong \rho_2 \quad \rho_2 \cong \rho_3}{\rho_1 \cong \rho_3}$$

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma;\Sigma\ \vdash\ x:\tau}$$

$$(abs)\quad \frac{\Gamma,x:\tau;\Sigma\ \vdash\ U:\tau'}{\Gamma;\Sigma\ \vdash\ (\lambda x^{\tau}.U):\tau\to\tau'}$$

$$(app)\quad \frac{\Gamma;\Sigma\ \vdash\ U:\tau\to\tau' \qquad \Gamma;\Sigma\ \vdash\ V:\tau}{\Gamma;\Sigma\ \vdash\ UV:\tau'}$$

$$(hvar)\quad \frac{(h:[\vec{\tau}]\tau)\in\Sigma}{\Gamma;\Sigma\ \vdash\ h:[\vec{\tau}]\tau}$$

$$(mabs)\quad \frac{\Gamma,\vec{x}:\vec{\tau};\Sigma\ \vdash\ U:\tau}{\Gamma;\Sigma\ \vdash\ (\Lambda\vec{x}^{\vec{\tau}}.U):[\vec{\tau}]\tau}$$

$$(mapp)\quad \frac{\Gamma;\Sigma\ \vdash\ U:[\vec{\tau}]\tau \qquad \Gamma;\Sigma\ \vdash\ \vec{V}:\vec{\tau}}{\Gamma;\Sigma\ \vdash\ U\langle\vec{V}\rangle:\tau}$$

$$(habs)\quad \frac{\Gamma;\Sigma,h:[\vec{\tau}]\tau\ \vdash\ U:\tau'}{\Gamma;\Sigma\ \vdash\ (\delta h^{[\vec{\tau}]\tau}.U):[\vec{\tau}]\tau\Rightarrow\tau'}$$

$$(fill)\quad \frac{\Gamma;\Sigma\ \vdash\ U:[\vec{\tau}]\tau\Rightarrow\tau' \qquad \Gamma;\Sigma\ \vdash\ V:[\vec{\tau}]\tau}{\Gamma;\Sigma\ \vdash\ U\lceil V\rceil:\tau'}$$

$$(cong)\quad \frac{\Gamma;\Sigma\ \vdash\ U:\rho \qquad \rho\cong\rho'}{\Gamma;\Sigma\ \vdash\ U:\rho'}$$

Figure 4.6: Type system for $\lambda c^{\cong}$

Example 4.3.4 Some examples and non-examples of types that are equal modulo $\mathtt{t} \to \mathtt{t} \cong \mathtt{t}$ are

$$\begin{array}{rcl} ([\mathtt{t},\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t} \to \mathtt{t} & \cong & ([\mathtt{t},\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t} \\ ([\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) & \cong & ([\mathtt{t} \to \mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t} \to \mathtt{t} \\ \mathtt{t} \to ([\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t} & \ncong & ([\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t}. \end{array}$$

With the congruence defined on types, the types are divided into equivalence classes. Therefore, it is convenient to have a notion of a representative of such a class, and the best candidate for such a representative is a minimal one, that is, the type with the minimal number of $\mathtt{t}$'s.

Definition 4.3.5 (Minimal types) Let $\rho \in \mathcal{P}_{\cong}$. Then ρ is called minimal if for all $\rho' \in \mathcal{P}_{\cong}$ with $\rho \cong \rho'$, the number of $\mathtt{t}$'s in ρ is less than or equal to the number of $\mathtt{t}$'s in ρ'.

If ρ is minimal, than all its subtypes are minimal. Moreover, minimal types are unique in their equivalence classes, as is shown by the next lemma.

Lemma 4.3.6 *Let $\rho \in \mathcal{P}_{\cong}$. Then there is a unique minimal $\rho' \in \mathcal{P}_{\cong}$ with $\rho \cong \rho'$.*

Proof: First of all, such ρ can be reduced to minimal number of symbols repeatedly replacing $\mathtt{t} \to \mathtt{t}$ by $\mathtt{t}$. This procedure terminates because by each such replacement the type has one $\mathtt{t}$ less. Suppose there are two minimal types ρ_1 and ρ_2 of U. By induction to the sum of the number of symbols in the types one easily checks that these must be equal $\rho_1 = \rho_2$. The heart of the argument is that, if $\rho_1' \cong \rho_2'$, then the head constructors of these types must be the same, otherwise they are either not equal modulo $\mathtt{t} \to \mathtt{t} \cong \mathtt{t}$ or not minimal. QED

The typing rules of $\lambda c^{\cong}$ are the typing rules of $\lambda c^{\to}$ plus the congruence rule, which introduces the congruence on types.

Definition 4.3.7 (Type system for $\lambda c^{\cong}$) A term $U \in \textsc{Ter}(\lambda c)$ is typable by ρ from the bases Γ, Σ, if $\Gamma; \Sigma \vdash U : \rho$ can be derived using the typing rules displayed in Figure 4.6.

Example 4.3.8 Examples of well-typed $\lambda c^{\cong}$-terms are (recall that, without type decorations, $\Omega \equiv (\lambda x.\, x\, x)(\lambda x.\, x\, x)$)

$$\begin{array}{l} \vdash \Omega : \mathtt{t} \\ \vdash \lambda c^{[\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}}.\, \lambda x^{\mathtt{t}}.\, c\lceil \Lambda y^{\mathtt{t}}.\, x \rceil \quad : ([\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t} \\ \vdash \lambda x^{\mathtt{t}}.\, \lambda c^{[\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}}.\, c\lceil \Lambda y^{\mathtt{t}}.\, x \rceil \quad : \mathtt{t} \to ([\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}) \to \mathtt{t} \\ \vdash \lambda x^{\mathtt{t}}.\, (\delta h^{[\mathtt{t}]\mathtt{t}}.\, \lambda z^{\mathtt{t}}.\, h\langle z \rangle)\lceil \Lambda y^{\mathtt{t}}.\, x \rceil \quad : \mathtt{t} \\ \vdash \lambda x^{\mathtt{t}}.\, x \quad : (\mathtt{t} \to \mathtt{t}) \to \mathtt{t} \\ \vdash \lambda x^{\mathtt{t}}.\, x \quad : \mathtt{t} \to \mathtt{t} \\ \vdash \lambda x^{\mathtt{t} \to \mathtt{t}}.\, x \quad : \mathtt{t}. \end{array}$$

Remark 4.3.9 There is also an alternative definition of such typing without congruence. Instead of defining types and then congruence on types, the definition of minimal types could have been used. The minimal types can be defined inductively as:

$$\begin{array}{lcll} \rho & = & \mathtt{t} & \\ \rho & = & \tau \to \sigma & \text{with } \tau,\sigma \text{ minimal and } \tau \ncong \mathtt{t} \text{ or } \sigma \ncong \mathtt{t} \\ \rho & = & [\vec{\tau}]\tau & \text{with } \vec{\tau} \text{ and } \tau \text{ minimal} \\ \rho & = & [\vec{\tau}]\tau \Rightarrow \sigma & \text{with } \vec{\tau}, \tau \text{ and } \sigma \text{ minimal.} \end{array}$$

In addition, the typing rules (*abs*) and (*app*) need to be split to deal with λ-terms and other terms, as shown below.

$$(abs_t)\ \frac{\Gamma, x:\mathtt{t}; \Sigma \vdash U:\mathtt{t}}{\Gamma;\Sigma \vdash \lambda x^{\mathtt{t}}.U:\mathtt{t}} \qquad (abs)\ \frac{\Gamma, x:\tau; \Sigma \vdash U:\tau'}{\Gamma;\Sigma \vdash \lambda x^{\tau}.U:\tau\to\tau'}$$

$$(app_t)\ \frac{\Gamma;\Sigma \vdash U:\mathtt{t} \quad \Gamma;\Sigma \vdash V:\mathtt{t}}{\Gamma;\Sigma \vdash UV:\mathtt{t}} \qquad (app)\ \frac{\Gamma;\Sigma \vdash U:\tau\to\tau' \quad \Gamma;\Sigma \vdash V:\tau}{\Gamma;\Sigma \vdash UV:\tau'}$$

In such a typing, the types in terms, as for example $\mathtt{t}$ in $\lambda x^{\mathtt{t}}.x$, are minimal; this is not the case in our definition of typing (see the last term in Example 4.3.8). However, we prefer our definition with less typing rules.

Definition 4.3.10 ($\lambda c^{\cong}$) The terms of $\lambda c^{\cong}$ are the well-typed terms of λc according to Definition 4.3.7. The rewrite rules are the rules (β), ($\textcircled{m}\beta$) and (*fill*) of λc, now restricted to $\lambda c^{\cong}$-terms.

i) The lambda calculus rewrite rule is:

$$(\lambda x^{\tau}.U)V \to U[\![x := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$\begin{array}{ll} (\Lambda \vec{x}^{\vec{\tau}}.U)\langle \vec{V}\rangle \to U[\![\vec{x} := \vec{V}]\!] & (\textcircled{m}\beta) \\ (\delta h^{[\vec{\tau}]\tau}.U)\lceil V \rceil \to U[\![h := V]\!]. & (\mathit{fill}) \end{array}$$

Definition 4.3.11 Let the ARS $\mathcal{A}_{\lambda c^{\cong}}$ be

$$\mathcal{A}_{\lambda c^{\cong}} = \langle \textsc{Ter}(\lambda c^{\cong}), \to_{\beta}, \to_{\textcircled{m}\beta}, \to_{\mathit{fill}} \rangle.$$

We call the ARS $\mathcal{A}_{\lambda c^{\cong}}$ the underlying ARS of the calculus $\lambda c^{\cong}$.

The calculus $\lambda c^{\cong}$ is a subsystem of the framework λc.

The proof that the calculus $\lambda c^{\cong}$ is a subsystem of λc is a standard one, via the Generation lemma, Substitution lemma and Subject reduction lemma. We only state these lemmas; the proofs are analogous to the proof of the same lemmas in the case of $\lambda c^{\rightarrow}$. The only extra difficulty is to make distinction between $\rho = \rho'$ and $\rho \cong \rho'$, when necessary. Furthermore, we also show that each $\lambda c^{\cong}$-term has a unique minimal type.

Lemma 4.3.12 (Generation lemma)

i) *If* $\Gamma;\Sigma \vdash u : \rho$ *then there is* ρ' *such that* $\rho \cong \rho'$ *and* $(u : \rho') \in \Gamma \cup \Sigma$.

ii) *If* $\Gamma;\Sigma \vdash \lambda x^{\rho'}.U : \rho$ *then* $\rho' \in \mathcal{T}_{\cong}$ *and there is a* $\tau \in \mathcal{T}_{\cong}$ *such that* $\rho \cong \rho' \to \tau$ *and* $\Gamma, x : \rho';\Sigma \vdash U : \tau$.

iii) *If* $\Gamma;\Sigma \vdash U_1 U_2 : \rho$ *then* $\rho \in \mathcal{T}_{\cong}$ *and there is a* $\tau \in \mathcal{T}_{\cong}$ *such that* $\Gamma;\Sigma \vdash U_1 : \tau \to \rho$ *and* $\Gamma;\Sigma \vdash U_2 : \tau$.

iv) *If* $\Gamma;\Sigma \vdash \Lambda\vec{x}^{\vec{\rho}}.U : \rho$ *then* $\vec{\rho} \in \mathcal{T}_{\cong}$, *there is a* $\tau \in \mathcal{T}_{\cong}$ *such that* $\rho \cong [\vec{\rho}]\tau$ *and* $\Gamma, \vec{x} : \vec{\rho};\Sigma \vdash U : \tau$.

v) *If* $\Gamma;\Sigma \vdash U\langle\vec{U}\rangle : \rho$ *then* $\rho \in \mathcal{T}_{\cong}$ *and there are* $\vec{\tau} \in \mathcal{T}_{\cong}$ *such that* $\Gamma;\Sigma \vdash U : [\vec{\tau}]\rho$ *and* $\Gamma;\Sigma \vdash \vec{U} : \vec{\tau}$.

vi) *If* $\Gamma;\Sigma \vdash \delta h^{\rho'}.U : \rho$ *then* $\rho' = [\vec{\tau}]\tau$ *and there is a* $\tau' \in \mathcal{T}_{\cong}$ *such that* $\rho \cong [\vec{\tau}]\tau \Rightarrow \tau'$ *and* $\Gamma;\Sigma, h : [\vec{\tau}]\tau \vdash U : \tau'$.

vii) *If* $\Gamma;\Sigma \vdash U_1\lceil U_2\rceil : \rho$ *then* $\rho \in \mathcal{T}_{\cong}$ *and there are* $\vec{\tau}, \tau \in \mathcal{T}_{\cong}$ *such that* $\Gamma;\Sigma \vdash U_1 : [\vec{\tau}]\tau \Rightarrow \rho$ *and* $\Gamma;\Sigma \vdash U_2 : [\vec{\tau}]\tau$.

Lemma 4.3.13 (Substitution lemma) *If* $\Gamma, \vec{u} : \vec{\rho}, \Sigma \vdash U : \rho$ *and* $\Gamma;\Sigma \vdash \vec{V} : \vec{\rho}$ *then* $\Gamma;\Sigma \vdash U[\![\vec{u} := \vec{V}]\!] : \rho$

Lemma 4.3.14 (Subject reduction) *If* $\Gamma;\Sigma \vdash U : \rho$ *and* $U \twoheadrightarrow V$, *then* $\Gamma;\Sigma \vdash V : \rho$.

Theorem 4.3.15 (Subsystem $\lambda c^{\cong}$) *The underlying ARS* $\mathcal{A}_{\lambda c^{\cong}}$ *of the calculus* $\lambda c^{\cong}$ *is an indexed sub-ARS of the underlying ARS* $\mathcal{A}_{\lambda c}$ *of the context calculus* λc.

Proof: One can easily check that the ARS $\mathcal{A}_{\lambda c^{\cong}}$ satisfies the conditions of the definition (Definition 1.1.14) of an indexed ARS of $\mathcal{A}_{\lambda c}$. QED

The terms of $\lambda c^{\cong}$ do not have a unique type, because of the congruence on types and the congruence typing rule. However, each $\lambda c^{\cong}$-term has a unique minimal type.

Lemma 4.3.16 *Let* U *be a* $\lambda c^{\cong}$*-term. Then* U *has a unique minimal type.*

Proof: Let U be a $\lambda c^{\cong}$-term. That means that there is a derivation of U, say $\Gamma;\Sigma \vdash U : \rho$. By Lemma 4.3.6, there is a unique minimal ρ' with $\rho \cong \rho'$. Then, by the congruence typing rule, $\Gamma;\Sigma \vdash U : \rho'$. QED

Properties of rewriting in $\lambda c^{\cong}$

Rewriting in $\lambda c^{\cong}$ is confluent. The proof is the same as in the case of $\lambda c^{\rightarrow}$ (see the proof of Theorem 4.2.10), which relies on the commutation property of every pair of the rewrite rules in λc.

Theorem 4.3.17 *The calculus $\lambda c^{\cong}$ is confluent.*

Rewriting in $\lambda c^{\cong}$ is not strongly, or even weakly, normalising, because the untyped λ-terms can be typed, including the notorious Ω. However, if the rewriting is limited to the rewriting which involves the context machinery, it is strongly normalising. We show this property into more detail. First we define the rewriting which is related to contexts as the rewriting which is not a pure β-step.

Definition 4.3.18 ($\rightarrow_{[]}$) The rewrite step $U \rightarrow V$ in $\lambda c^{\cong}$ is called a $[]$-rewrite step (notation $U \rightarrow_{[]} V$) if it is *not* the case that

$$U \equiv C[(\lambda x^{\tau}.\, M)\, N] \rightarrow_{\beta} C[M[\![x := N]\!]] \equiv V$$

with $\tau \cong \mathtt{t}$ and $M : \mathtt{t}$.

Let the restriction of $\lambda c^{\cong}$ to the rewriting related to contexts be called $\lambda c^{\cong}_{[]}$.

Definition 4.3.19 (The calculus $\lambda c^{\cong}_{[]}$) Let the calculus $\lambda c^{\cong}_{[]}$ be defined on the terms of λc^{λ} with the rewrite relations generated by $\rightarrow_{[]}$.

Remark 4.3.20 The set of terms $\textsc{Ter}(\lambda c^{\cong}_{[]})$ of the calculus $\lambda c^{\cong}_{[]}$ is closed under $\rightarrow_{[]}$. It is closed because $\textsc{Ter}(\lambda c^{\cong}_{[]}) = \textsc{Ter}(\lambda c^{\cong})$ and the set $\textsc{Ter}(\lambda c^{\cong})$ is closed under rewriting of $\rightarrow$, which includes $\rightarrow_{[]}$. Note that the calculus $\lambda c^{\cong}_{[]}$ is not a subsystem of the calculus $\lambda c^{\cong}$ in the sense that the underlying ARS of $\lambda c^{\cong}_{[]}$ is not an indexed sub-ARS of the underlying ARS $\mathcal{A}_{\lambda c^{\cong}}$ of $\lambda c^{\cong}$. The calculus is not a subsystem, because the rewrite relation $\rightarrow_{[]}$ is not the restriction of $\rightarrow_{\beta}$ to $\textsc{Ter}(\lambda c^{\cong}_{[]}) = \textsc{Ter}(\lambda c^{\cong})$. However, the fact that the calculus $\lambda c^{\cong}_{[]}$ is not a subsystem of $\lambda c^{\cong}$ does not disturb the proof of strong normalisation of $\rightarrow_{[]}$.

In fact, one could split the β-rewrite relation into two rewrite relations and then consider $\lambda c^{\cong}_{[]}$ as an indexed sub-ARS of $\lambda c^{\cong}$. Let

$$(\lambda x^{\tau}.\, M)\, N \rightarrow_{\beta} M[\![x := N]\!]$$
$$(\lambda x^{\sigma}.\, M')\, N' \rightarrow_{\beta[]} M'[\![x := N']\!]$$

with $\tau \cong \mathtt{t}$, $M : \mathtt{t}$, and either $\sigma \ncong \mathtt{t}$ or M' is not of type $\mathtt{t}$. Then

$$\mathcal{A}_{\lambda c^{\cong}_{[]}} = \langle \textsc{Ter}(\lambda c^{\cong}), \rightarrow_{\beta[]}, \rightarrow_{\textcircled{m}\beta}, \rightarrow_{\mathit{fill}} \rangle$$

is an indexed sub-ARS of $\mathcal{A}_{\lambda c^{\cong}}$.

The proof of strong normalisation with respect to $\rightarrow_{[]}$ proceeds as in the case of strong normalisation with respect to $\rightarrow_c$ in λc^{λ}. This proof is also an adaptation of Tait's method as presented in [HS86].

For the purpose of the proof we will first define the notion of strong computability with respect to $\rightarrow_{[]}$. Then we will show that it implies strong normalisation with respect to $\rightarrow_{[]}$ and that each $\lambda c^{\cong}$-term is strongly computable with respect to $\rightarrow_{[]}$. The definition and the proofs use minimal types.

Among other properties of the $\lambda c^{\cong}$-terms, the definition and the proofs below rely on the following one. If $U : \mathtt{t}$ and U is $SN_{[]}$, then for all V which are $SC_{[]}$, it holds that UV is $SC_{[]}$. The application is $SC_{[]}$ because no $[]$-redexes are created by the formation of UV; hence, UV is $SN_{[]}$.

Notation. If U is strongly normalising with respect to $\rightarrow_{[]}$, then we will say U is $SN_{[]}$. Analogously, if U is strongly computable with respect to $\rightarrow_{[]}$, we will say U is $SC_{[]}$

Definition 4.3.21 (Strong computability in $\lambda c^{\cong}$) Strong computability of $\lambda c^{\cong}$-terms is defined by induction to the minimal types of terms:

i) Let $U : \mathtt{t}$. The term U is strongly computable with respect to $\rightarrow_{[]}$ if and only if U is strongly normalising with respect to $\rightarrow_{[]}$.

ii) Let $U : \tau \rightarrow \sigma$ with $\tau \rightarrow \sigma$ a minimal type. Then, $\tau \ncong \mathtt{t}$ or $\sigma \ncong \mathtt{t}$. The term U is $SC_{[]}$ if and only if for all strongly computable terms V with $V : \tau$ the term UV (of type σ) is $SC_{[]}$.

iii) Let $U : [\vec{\tau}]\tau$ with $[\vec{\tau}]\tau$ a minimal type. The term U is $SC_{[]}$ if and only if for all strongly computable terms $\vec{V}$ with $V_i : \tau_i$ for $1 \leq i \leq |\vec{V}|$ the term $U\langle\vec{V}\rangle$ (of type τ) is $SC_{[]}$.

iv) Let $U : [\vec{\tau}]\tau \Rightarrow \sigma$ with $[\vec{\tau}]\tau \Rightarrow \sigma$ a minimal type. The term U is $SC_{[]}$ if and only if for all strongly computable terms V with $V : [\vec{\tau}]\tau$ the term $U\lceil V\rceil$ (of type σ) is $SC_{[]}$.

Notation. For the sake of short notation, we introduce a notation for any applicator and a notation for any type constructor. Let $\cdot$ denote any application, that is, the implicit application $\cdot$ itself, the multiple application $\langle\rangle$ and the hole filling $\lceil\ \rceil$. Let $\rightsquigarrow$ denote any of the type constructors, $\rightarrow$, $\Rightarrow$ or $[\,]$. That is, instead of $\tau \rightarrow \sigma$ we will write $\tau \rightsquigarrow \sigma$ in which τ and σ have also been adapted, instead of $\rho \Rightarrow \tau$ we will write $\rho \rightsquigarrow \tau$ with adapted ρ and τ, and instead of $[\vec{\tau}]\tau$ we will write $\tau_1 \rightsquigarrow \ldots \rightsquigarrow \tau_n \rightsquigarrow \tau$ with adapted $\vec{\tau}$ and τ. In the rest of the section we work only on typable terms and on minimal types.

Lemma 4.3.22 *Let ρ be a minimal type.*

i) Let $\vec{U}$ be strongly normalising with respect to $\rightarrow_{[]}$. Let $u \cdot \vec{U} : \rho$. Then $u \cdot \vec{U}$ is strongly computable.

ii) Let U be a $\lambda c^{\cong}$-term of type ρ. If U is $SC_{[]}$ then U is $SN_{[]}$.

Proof: Let $\vec{U}$ be $SN_{[]}$. Let $u \cdot \vec{U} : \rho$, with ρ a minimal type. The two statements are proved simultaneously by induction to ρ.

$\rho = \mathtt{t}$**:**

i) Then $u \cdot \vec{U}$ is $SN_{[]}$, and by definition it is also $SC_{[]}$.

ii) If $U : \mathtt{t}$ is $SC_{[]}$, then it is $SN_{[]}$ by definition.

$\rho = \tau \rightarrow \sigma$**:**

i) Let $W : \tau$ be $SC_{[]}$. By the induction hypothesis for (ii), W is also $SN_{[]}$. Then $(u \cdot \vec{U})W$ is also $SC_{[]}$ by the induction hypothesis of (i). Then by definition, $u \cdot \vec{U}$ is $SC_{[]}$.

ii) Let $U : \rho$ be $SC_{[]}$. Let $x : \tau$ be a fresh variable. Then, by the induction hypothesis for (i), x is $SC_{[]}$. By definition Ux is $SC_{[]}$. By the induction hypothesis for (ii), the term Ux is $SN_{[]}$. Then the subterm U is $SN_{[]}$ too.

$\rho = [\vec{\tau}]\tau$**:**

i) Let $\vec{W} : \vec{\tau}$ be $SC_{[]}$. Then, by the induction hypothesis for (ii), $\vec{W}$ are also $SN_{[]}$. Then $(u \cdot \vec{U})\langle\vec{W}\rangle$ is $SC_{[]}$ by the induction hypothesis for (i). Then by definition, $u \cdot \vec{U}$ is $SC_{[]}$.

ii) Let $U : \rho$ be $SC_{[]}$. Let $\vec{x} : \vec{\tau}$ be fresh variables. Then, by the induction hypothesis for (i), $\vec{x}$ are $SC_{[]}$. Then by definition, $U\langle\vec{x}\rangle$ is $SC_{[]}$. By the induction hypothesis for (ii), $U\langle\vec{x}\rangle$ is $SN_{[]}$. Then its subterm U must be $SN_{[]}$ too.

$\rho = [\vec{\tau}]\tau \Rightarrow \sigma$**:**

i) Let $W : [\vec{\tau}]\tau$ be $SC_{[]}$. Then, by the induction hypothesis for (ii), W is also $SN_{[]}$. Then $(u \cdot \vec{U})\lceil W \rceil$ is $SC_{[]}$ by the induction hypothesis for (i). Then by definition, $u \cdot \vec{U}$ is $SC_{[]}$.

ii) Let $U : \rho$ be $SC_{[]}$. Let $h : [\vec{\tau}]\tau$ be a fresh variable. Then, by the induction hypothesis for (i), h is $SC_{[]}$. Then by definition, $U\lceil h \rceil$ is $SC_{[]}$. By the induction hypothesis for (ii), $U\lceil h \rceil$ is $SN_{[]}$. Then its subterm U must be $SN_{[]}$ too.

QED

Lemma 4.3.23

i) *If* $U[\![x := V]\!]$ *is* $SC_{[]}$, *then* $(\lambda x^{\tau}.U)V$ *is* $SC_{[]}$.

ii) *If* $U[\![\vec{x} := \vec{V}]\!]$ *is* $SC_{[]}$, *then* $(\Lambda \vec{x}^{\vec{\tau}}.U)\langle \vec{V} \rangle$ *is* $SC_{[]}$.

iii) *If* $U[\![h := V]\!]$ *is* $SC_{[]}$, *then* $(\delta h^{[\vec{\tau}]\tau}.U)\lceil V \rceil$ *is* $SC_{[]}$.

Proof:

i) Let $U[\![v := V]\!]$ be $SC_{[]}$. Let $U[\![v := V]\!] : \rho_1 \rightsquigarrow \ldots \rightsquigarrow \rho_n \rightsquigarrow \mathtt{t}$ with $\rho_1 \rightsquigarrow \ldots \rightsquigarrow \rho_n \rightsquigarrow \mathtt{t}$ minimal. Let $\vec{W}$ be $SC_{[]}$ with $W_i : \rho_i$ for $1 \leq i \leq n$. Then $U\,[\![v := V]\!] \cdot \vec{W}$ is $SC_{[]}$ and of type $\mathtt{t}$. Therefore it is $SN_{[]}$ by definition.

Suppose $((\lambda x^{\tau}.U)V) \cdot \vec{W}$ has an infinite reduction: Then, since $U, V, \vec{W}$ are $SN_{[]}$, eventually the head redex has to be reduced:

$$((\lambda x^{\tau}.U)V) \cdot \vec{W} \twoheadrightarrow_{[]} ((\lambda x^{\tau}.U')V') \cdot \vec{W}' \rightarrow_{\beta} U'\,[\![x := V']\!] \cdot \vec{W}' \twoheadrightarrow_{[]} \ldots$$

Then $U\,[\![x := V]\!] \cdot \vec{W}$ has an infinite reduction too:

$$U\,[\![x := V]\!] \cdot \vec{W} \twoheadrightarrow_{c} U'\,[\![x := V']\!] \cdot \vec{W}' \twoheadrightarrow_{c} \ldots$$

This is a contradiction with the fact that $U\,[\![x := V]\!] \cdot \vec{W}$ is $SN_{[]}$.

Then, $((\lambda x^{\tau}.U)V) \cdot \vec{W}$ is $SN_{[]}$. Because this term is of type $\mathtt{t}$, it is $SC_{[]}$ by the definition of $SC_{[]}$. Because $\vec{W}$ are arbitrary $SC_{[]}$ terms, $(\lambda x^{\tau}.U)V$ is $SC_{[]}$ by definition.

ii) Analogously.

iii) Analogously. QED

Lemma 4.3.24 *Let* U *be a* $\lambda c^{\cong}$*-term. Then* U *is* $SC_{[]}$.

Proof: The proof is conducted by induction to U. For the induction step to work, we need to prove the following strengthening of the lemma:

> Let $\vec{u}$ be $|\vec{V}|$ variables such that u_i and V_i are of the same type for $1 \leq i \leq |\vec{V}|$. If $\vec{V}$ are $SC_{[]}$ then $U[\![\vec{u} := \vec{V}]\!]$ is $SC_{[]}$.

So let $\vec{u}$ and $\vec{V}$ be as required. Let $W^{\star}$ denote $W[\![\vec{u} := \vec{V}]\!]$. We treat only three cases:

$U \equiv u$**:** with $u \not\equiv u_i$ for all $1 \leq i \leq |\vec{u}|$.

Then $U^{\star} = u$. By Lemma 4.3.22(i), u is $SC_{[]}$.

$U \equiv W_1 W_2$**:** Then $U^\star = W_1^\star W_2^\star$. By the induction hypothesis, the terms $W_1^\star$ and $W_2^\star$ are $SC_{[]}$.

If $W_1 : \mathtt{t}$ and $W_2 : \mathtt{t}$, then $U : \mathtt{t}$. Because $W_1^\star$ and $W_2^\star$ are $SC_{[]}$ of type $\mathtt{t}$, they are also $SN_{[]}$ by definition. Then $W_1^\star W_2^\star$ is also $SN_{[]}$ because no $[]$-redexes are created by the application. Because $W_1^\star W_2^\star$ is of type $\mathtt{t}$ it is also $SC_{[]}$ by definition.

Otherwise, because $W_1^\star$ and $W_2^\star$ are $SC_{[]}$, so is $W_1^\star W_2^\star$.

$U \equiv \delta h^{[\vec{\tau}]\tau}.W$**:** Then $U^\star = \delta h^{[\vec{\tau}]\tau}.W^\star$. Let W' be a $SC_{[]}$ terms of the same type as h. Then $W\,[\![\vec{u} := \vec{V}]\!]\,[\![h := W']\!]$ is $SC_{[]}$, by the induction hypothesis. That is, $W^\star[\![h := W']\!]$ is $SC_{[]}$. By Lemma 4.3.23(iii), the term $(\delta h^{[\vec{\tau}]\tau}.W^\star)\lceil W' \rceil$ is $SC_{[]}$. By the definition of $SC_{[]}$, the term $\delta h^{[\vec{\tau}]\tau}.W^\star$ is $SC_{[]}$.

QED

Theorem 4.3.25 (Strong normalisation of $\rightarrow_{[]}$) *The rewriting with respect to $\rightarrow_{[]}$ in $\lambda c^{\cong}$ is strongly normalising.*

Proof: This theorem is a corollary of Lemma 4.3.22(ii) and Lemma 4.3.24. QED

Theorem 4.3.26 *The rewriting with respect to $\rightarrow_{[]}$ in $\lambda c^{\cong}$ is complete.*

Proof: By Theorem 4.3.25 the rewriting with respect to $\rightarrow_{[]}$ in $\lambda c^{\cong}$ is strongly normalising. Furthermore, by keeping in mind that rewriting with respect to $\rightarrow_{[]}$ preserves types, one can easily check that the rewriting with respect to $\rightarrow_{[]}$ is weakly confluent. Then, by Newman's lemma 1.1.11, the rewriting with respect to $\rightarrow_{[]}$ is also confluent. QED

Comparison to the calculi λc^{λ} and $\lambda c^{\rightarrow}$

We compare $\lambda c^{\cong}$ to the calculi λc^{λ} and $\lambda c^{\rightarrow}$. We do so by defining translations of these calculi into $\lambda c^{\cong}$ in such a way that rewrite steps are preserved. Furthermore, we will show that the calculus $\lambda c^{\cong}$ is more expressive than λc^{λ} and $\lambda c^{\rightarrow}$, because it contains representations of objects from lambda calculus which cannot be represented in λc^{λ} or $\lambda c^{\rightarrow}$.

The calculus λc^{λ} can be translated into $\lambda c^{\cong}$ rather straightforwardly. Translation involves linearisation of the simultaneous hole filling of λc^{λ} and translation of composition into a definable function on contexts.

Definition 4.3.27 (Translation of λc^{λ} into $\lambda c^{\cong}$)

i) The types of λc^{λ} are translated to $\lambda c^{\cong}$ as follows:

$$
\begin{array}{lcl}
[\![\ \mathtt{t}\ ]\!] & = & \mathtt{t} \\
[\![[\vec{\mathtt{t}}]\ \mathtt{t}\ ]\!] & = & [\vec{\mathtt{t}}]\mathtt{t} \\
[\![[\vec{\mathtt{t}}_1]\ \mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}]\!] & = & [\vec{\mathtt{t}}_1]\mathtt{t} \Rightarrow \ldots \Rightarrow [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} \\
[\![[\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\ \mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t})]\!] & = & [\vec{\mathtt{t}}]([\vec{\mathtt{t}}_1]\mathtt{t} \Rightarrow \ldots \Rightarrow [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t}).
\end{array}
$$

ii) The terms of λc^{λ} are translated to $\lambda c^{\cong}$ as follows:

$$\begin{array}{lcll}
\llbracket u \rrbracket & = & u & \\
\llbracket\ \lambda x.U\ \rrbracket & = & \lambda x^{\mathtt{t}}.\llbracket U \rrbracket & \\
\llbracket\ UV\ \rrbracket & = & \llbracket U \rrbracket \llbracket V \rrbracket & \\
\llbracket\ \Lambda \vec{x}.U\ \rrbracket & = & \Lambda \vec{x}^{\vec{\mathtt{t}}}.\llbracket U \rrbracket & \\
\llbracket\ U\langle \vec{V} \rangle\ \rrbracket & = & \llbracket U \rrbracket \langle \llbracket \vec{V} \rrbracket \rangle & \\
\llbracket\ \delta \vec{h}.U\ \rrbracket & = & \delta h_1^{[\vec{\mathtt{t}}_1]\mathtt{t}}. \ldots \delta h_n^{[\vec{\mathtt{t}}_n]\mathtt{t}}.\llbracket U \rrbracket & \text{if } U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} \\
\llbracket\ U\lceil \vec{V} \rceil\ \rrbracket & = & \llbracket U \rrbracket \lceil \llbracket V_1 \rrbracket \rceil \ldots \lceil \llbracket V_n \rrbracket \rceil & \\
\llbracket\ U \circ \vec{V}\ \rrbracket & = & \underline{comp}\ \llbracket U \rrbracket \llbracket \vec{V} \rrbracket &
\end{array}$$

where, if

$$\begin{array}{l}
U : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} \\
V_i : [\vec{\mathtt{t}}_i]([\vec{\mathtt{t}}_{i,1}]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{i,l_i}]\mathtt{t} \Rightarrow \mathtt{t}) \quad (1 \leq i \leq n)
\end{array}$$

then

$$\begin{array}{lcl}
\underline{comp} & \equiv & \lambda u.\, \delta v_1. \ldots \delta v_n.\, \delta h_{1,1}. \ldots \delta h_{n,l_n}. \\
& & \quad u \lceil \Lambda \vec{x}_1.\, (v_1 \langle \vec{x}_1 \rangle) \lceil h_{1,1} \rceil \ldots \lceil h_{1,l_1} \rceil \rceil \ldots \\
& & \quad \lceil \Lambda \vec{x}_n.\, (v_n \langle \vec{x}_n \rangle) \lceil h_{n,1} \rceil \ldots \lceil h_{n,l_n} \rceil \rceil
\end{array}$$

where the types of the abstractions, which have been left out for the sake of readability, are

$$\begin{array}{lll}
u & : [\vec{\mathtt{t}}_1]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_n]\mathtt{t} \Rightarrow \mathtt{t} & \\
v_i & : [\vec{\mathtt{t}}_i]([\vec{\mathtt{t}}_{i,1}]\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}_{i,l_i}]\mathtt{t} \Rightarrow \mathtt{t}) & (1 \leq i \leq n) \\
h_{i,j} & : [\vec{\mathtt{t}}_{i,j}]\mathtt{t} & (1 \leq i \leq n, 1 \leq j \leq l_i) \\
\vec{x}_i & : \vec{\mathtt{t}}_i & (1 \leq i \leq n).
\end{array}$$

iii) Let Δ be a basis. Then $\llbracket \Delta \rrbracket = \{(u : \llbracket \rho \rrbracket) \mid (u : \rho) \in \Delta\}$.

Translation preserves typing.

Proposition 4.3.28 *If* $\Gamma; \Sigma \vdash_{\lambda c^{\lambda}} U : \rho$, *then* $\llbracket \Gamma \rrbracket; \llbracket \Sigma \rrbracket \vdash_{\lambda c^{\cong}} \llbracket U \rrbracket : \llbracket \rho \rrbracket$.

Proof: By induction to the length of the derivation $\Gamma; \Sigma \vdash_{\lambda c^{\lambda}} U : \rho$. One checks that each derivation rule of λc^{λ} can be translated into a derivation in $\lambda c^{\cong}$. QED

Translation preserves rewrite steps.

Proposition 4.3.29 *If* $U \to V$ *in* λc^{λ}, *then* $\llbracket U \rrbracket \twoheadrightarrow \llbracket V \rrbracket$ *in* $\lambda c^{\cong}$.

Proof: By induction to U. In the base cases, check that a reduct in λc^{λ} can be translated to a reduct in $\lambda c^{\cong}$. Note that the translation of one step in λc^{λ} may lead to many (including zero) steps in $\lambda c^{\cong}$. For example, $(\delta\epsilon.U)\lceil\rceil \rightarrow_{\mathit{fill}} U$ is translated into an empty rewrite sequence. QED

Note that the strong normalisation of the context-related rewriting in λc^{λ} follows from the strong normalisation of $\rightarrow_{[]}$ rewriting in $\lambda c^{\cong}$.

The calculus $\lambda c^{\cong}$ is more expressive than λc^{λ}. We have, for example,

$$\vdash \quad (\lambda c^{[]\mathtt{t}\Rightarrow\mathtt{t}}.\,\lambda x^{\mathtt{t}}.\,c\lceil\Lambda\epsilon.\,x\rceil) \quad : \quad (([]\mathtt{t}\Rightarrow\mathtt{t})\rightarrow\mathtt{t}\rightarrow\mathtt{t}).$$

Note that the term $\lambda c^{[]\mathtt{t}\Rightarrow\mathtt{t}}.\,\lambda x^{\mathtt{t}}.\,c\lceil\Lambda\epsilon.\,x\rceil$ represents a function ranging over a context and a term which fills the hole of the context by the term without any communication. However, such an object cannot be represented in λc^{λ} because λc^{λ} does not support functions ranging over contexts.

Translation of $\lambda c^{\rightarrow}$ into $\lambda c^{\cong}$ is given next. The properties of the translation are proved in the similar way as in the case of the translation of λc^{λ} into $\lambda c^{\cong}$; we leave the proofs out.

Definition 4.3.30 (Translation of $\lambda c^{\rightarrow}$ into $\lambda c^{\cong}$)

i) The types of $\lambda c^{\rightarrow}$ are translated to $\lambda c^{\cong}$ as follows:

$$\begin{array}{lcl}
\llbracket\ \mathtt{a}\ \rrbracket & = & \mathtt{t} \\
\llbracket[\vec{\tau}]\tau\rrbracket & = & [\llbracket\vec{\tau}\rrbracket]\llbracket\tau\rrbracket \\
\llbracket[\vec{\tau}]\tau\Rightarrow\sigma\rrbracket & = & [\llbracket\vec{\tau}\rrbracket]\llbracket\tau\rrbracket\Rightarrow\llbracket\sigma\rrbracket \\
\llbracket[\vec{\sigma}]([\vec{\tau}]\tau\Rightarrow\sigma)\rrbracket & = & [\llbracket\vec{\sigma}\rrbracket]([\llbracket\vec{\tau}\rrbracket]\llbracket\tau\rrbracket\Rightarrow\llbracket\sigma\rrbracket).
\end{array}$$

ii) The terms of $\lambda c^{\rightarrow}$ are translated to $\lambda c^{\cong}$ as follows:

$$\begin{array}{lcl}
\llbracket\ u\rrbracket & = & u \\
\llbracket\ \lambda x^{\tau}.U\ \rrbracket & = & \lambda x^{\llbracket\tau\rrbracket}.\llbracket U\rrbracket \\
\llbracket\ UV\ \rrbracket & = & \llbracket U\rrbracket\llbracket V\rrbracket \\
\llbracket\ \Lambda\vec{x}^{\vec{\tau}}.U\ \rrbracket & = & \Lambda\vec{x}^{\llbracket\vec{\tau}\rrbracket}.\llbracket U\rrbracket \\
\llbracket\ U\langle\vec{V}\rangle\ \rrbracket & = & \llbracket U\rrbracket\langle\llbracket\vec{V}\rrbracket\rangle \\
\llbracket\ \delta h^{\rho}.U\ \rrbracket & = & \delta h^{\llbracket\rho\rrbracket}.\llbracket U\rrbracket \\
\llbracket\ U\lceil V\rceil\ \rrbracket & = & \llbracket U\rrbracket\lceil\llbracket V\rrbracket\rceil.
\end{array}$$

iii) Let Δ be a basis. Then $\llbracket\Delta\rrbracket = \{(u:\llbracket\rho\rrbracket) \mid (u:\rho)\in\Delta\}$.

Proposition 4.3.31 *If* $\Gamma;\Sigma\vdash_{\lambda c^{\rightarrow}} U:\rho$, *then* $\llbracket\Gamma\rrbracket;\llbracket\Sigma\rrbracket\vdash_{\lambda c^{\cong}}\llbracket U\rrbracket:\llbracket\rho\rrbracket$.

Proposition 4.3.32 *If* $U\rightarrow V$ *in* $\lambda c^{\rightarrow}$, *then* $\llbracket U\rrbracket\rightarrow\llbracket V\rrbracket$ *in* $\lambda c^{\cong}$.

The calculus $\lambda c^{\cong}$ is more expressive than the calculus $\lambda c^{\rightarrow}$, because in $\lambda c^{\cong}$ the terms of untyped lambda calculus can be represented, which is not the case in $\lambda c^{\rightarrow}$. For example, $\vdash_{\lambda c^{\cong}}\Omega:\mathtt{t}$.

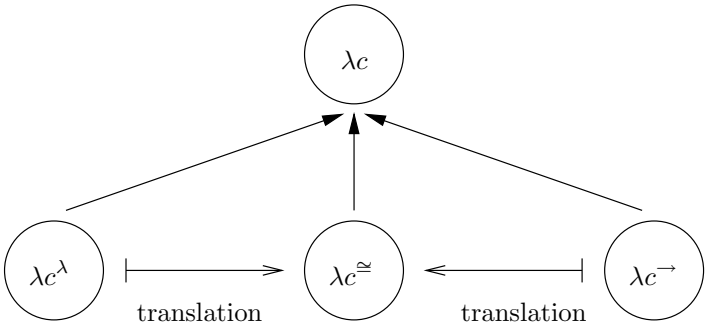


Figure 4.7: Relationship between λc and its applications

The introductory example in $\lambda c^{\cong}$

As a final remark, the introductory example 2.2.1 can be translated into $\lambda c^{\cong}$. The translations and the repaired commutation diagram has the same form as in Example 2.5.3 and in the case of λc^{λ}, now with type decorations:

$$\begin{array}{ccc} (\delta g^{[\mathtt{t}]\mathtt{t}}.(\lambda y^{\mathtt{t}}.g\langle y\rangle)x)\lceil\Lambda y^{\mathtt{t}}.xy\rceil & \to_{\beta} & (\delta g^{[\mathtt{t}]\mathtt{t}}.g\langle x\rangle)\lceil\Lambda y^{\mathtt{t}}.xy\rceil \\ \downarrow_{\mathit{fill}} & & \downarrow_{\mathit{fill}} \\ & & \downarrow_{\textcircled{\scriptsize m}\beta} \\ (\lambda y^{\mathtt{t}}.(\Lambda y^{\mathtt{t}}.xy)\langle y\rangle)x & \twoheadrightarrow_{\beta,\textcircled{\scriptsize m}\beta} & xx. \end{array}$$

Here, $\twoheadrightarrow_{\beta,\textcircled{\scriptsize m}\beta}$ stands for $\to_{\beta};\to_{\textcircled{\scriptsize m}\beta}$ or $\to_{\textcircled{\scriptsize m}\beta};\to_{\beta}$.

4.4 Summary of comparisons

In this section we summarise the relationship between the context calculi considered in this chapter.

See Figure 4.7. This figure shows the relationship between the calculi λc^{λ}, $\lambda c^{\to}$, $\lambda c^{\cong}$ and the framework λc. In the figure, the solid-line arrows denote the subsystem-relation in the sense of Definitions 1.1.13 and 1.1.14: $\mathcal{B} \to \mathcal{A}$ means that the underlying ARS of $\mathcal{B}$ is an (indexed) sub-ARS of the underlying ARS of $\mathcal{A}$. The $\mapsto$-arrows denote translation: $\mathcal{B} \mapsto \mathcal{A}$. By translation we mean that the elements of $\mathcal{B}$ can be translated into the elements of $\mathcal{A}$. Moreover, translation preserves rewrite steps.

We comment on the arrows:

- The calculi λc^{λ}, $\lambda c^{\to}$, and $\lambda c^{\cong}$ are all (indexed) subsystems of the framework λc. By ignoring the indices, they are actually full subsystems of the framework.

calculus	formalises contexts of	functions over contexts?	translated to
λc^{λ}	untyped λ-calculus	no	untyped λ-calculus
$\lambda c^{\rightarrow}$	$\lambda^{\rightarrow}$	yes	$\lambda^{\rightarrow}$
$\lambda c^{\cong}$	untyped λ-calculus	yes	untyped λ-calculus

Table 4.1: The expressivity of the context calculi

- The calculi λc^{λ} and $\lambda c^{\rightarrow}$ can be translated into the calculus $\lambda c^{\cong}$.

See Table 4.1. The table compares the expressivity of the calculi λc^{λ}, $\lambda c^{\rightarrow}$, and $\lambda c^{\cong}$ as context calculi, and their relationship with lambda calculi.

Chapter 5

De Bruijn's segments

In [Bru78], N.G. de Bruijn introduced a lambda calculus extended with contexts of a special form, called *segments*. The purpose of segments was facilitating definitions and manipulation of abbreviations in Automath (see [NGdV94]). The segment calculus included means for representing segments, variables ranging over segments and abstractions ranging over segments. In [Bal86, Bal87] H. Balsters gave a simply typed version of the segment calculus and proved confluence and subject reduction. Later, N.G. de Bruijn [Bru91] showed how telescopes, a special kind of segments, can be considered in a more general setting.

Segments are contexts, considering the structure of segments and transformations defined on segments. Technically, segments can be characterised as contexts with precisely one hole that is placed at a designated position. More precisely, the hole is placed at the leftmost innermost position of a segment, that is, the hole is the leftmost leaf in the tree representation of a segment.

De Bruijn's segment calculus takes a special place in this thesis. De Bruijn's segment calculus was the starting point of our research. A slight generalisation of these contexts has led to the framework λc.

In this chapter we concentrate on segments and the segment calculus as described in [Bru78] and [Bal87]. This chapter is organised as follows. Section 5.1 is an introduction to segments and de Bruijn's segment calculus. In the formalisation of segments and in designing a segment calculus, de Bruijn faced the same problems as one does when formalising λ-contexts: variable capturing and the problem of extending the β-rewrite relation to segments. However, de Bruijn solved these problems in a way that greatly differs from our approach in λc. De Bruijn's solution is also explained in this section.

Section 5.2 presents the calculus λc^s which is another application of the framework λc, in addition to the calculi presented in Chapter 4. From the point of view of lambda calculus, the calculus λc^s is an extension of the simply typed lambda calculus with polymorphic segments. The calculus λc^s is complete, that is, it has the confluence property and the strong normalisation property, which we will show here. Moreover, we will compare it to the calculus of de Bruijn and the calculus $\lambda c^{\rightarrow}$

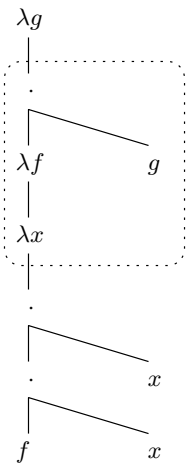


Figure 5.1: A segment of a λ-term

of Section 4.2, which also deals with simply typed lambda calculus with contexts.

5.1 De Bruijn's segments and segment calculus

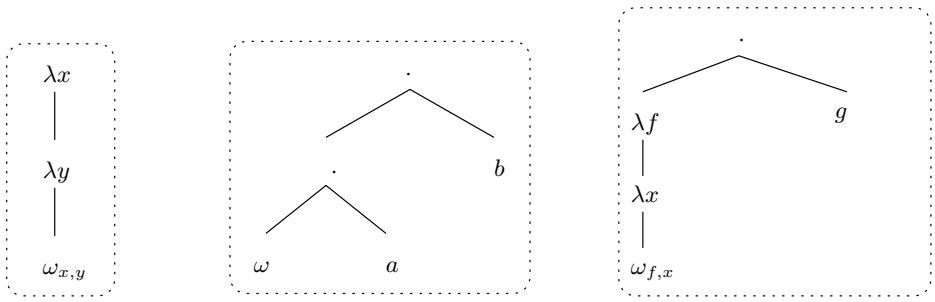
In this section we give a description of segments and de Bruijn's segment calculus. We also give a motivation for introducing segments and a short historical background. The notation and terminology used here are conform to the notation in [Bru78]. Moreover, we assume the reader is familiar with name-free notation for terms (also called de Bruijn notation, see for example [Bru72]) and an explicit substitution calculus (see for example [ACCL91] or [Ros96]).

The subsection on details of de Bruijn's calculus is rather technical, and it is meant as a comment on the first six pages of [Bru78].

Segments

Segments were introduced for abbreviating segments of λ-terms. An example of what is meant by a segment of a λ-term is given in Figure 5.1. In the figure the λ-term $\lambda g.(\lambda f.\lambda x.fxx)g$ is depicted in tree notation. The segment $(\lambda f.\lambda x.\_)g$ of the λ-term is included in a box. This segment is not a λ-term, because the 'body' of the abstractor λx is 'open'. Hence, the existing abbreviation mechanism (cf. Remark 1.2.16) in λ-calculus using (term) variables cannot be employed for segments.

In the context of the project Automath de Bruijn extended λ-calculus with a notion of segment. A segment is, in the words of de Bruijn (see [Bru78], p.20), a λ-tree which is open-ended at the end of the spine. The spine of a λ-tree is the path from the root of the λ-tree down to the leftmost leaf of the λ-tree. Thus, the end of the spine of a λ-tree is the leftmost leaf of that λ-tree (cf. 'heart' position in [Geu93]).



a) $\lambda x.\, \lambda y.\, \omega_{x,y}$ b) $\omega\, a\, b$ c) $(\lambda f.\, \lambda x.\, \omega_{f,x})\, g$

Figure 5.2: Segments

By open-ended de Bruijn means that a segment is an incomplete λ-term: a segment becomes a λ-term upon filling the open end by an arbitrary λ-term. In de Bruijn's calculus, the open end of a segment is explicitly denoted by ω_L, where the label L denotes the binders in whose scope the open end is. The label is necessary, as we will see later, for establishing the communication between the segment and the λ-terms or segments to be put into ω.

Example 5.1.1 (Segments) Examples of segments are $\lambda x.\, \lambda y.\, \omega_{x,y}$, $\omega\, a\, b$, and $(\lambda f.\, \lambda x.\, \omega_{f,x})\, g$. These segments are depicted in tree notation in Figure 5.2. The label of ω in, for example, $(\lambda f.\, \lambda x.\, \omega_{f,x})\, g$ is f, x because ω lies in the scope of the binders λf and λx.

Notation. Because tree notation is in this thesis considered as an alternative notation, we will often confuse a λ-term or a segment and its tree representation. Sometimes we will use both notations in one structure.

As already suggested, the open end of a segment is intended to be filled. By filling the open end of a segment we mean literally replacing the open end ω by a λ-term or a segment. Filling the open end by a λ-term results in a λ-term. Moreover, the open end may also be filled by a segment to form a new, bigger segment. In both cases, upon filling the open end, variable capturing may occur: some free variables of the object filled into the open end may become bound by the binders of the segment. This variable capturing is, as in the case of filling holes of a context, intentional.

Example 5.1.2 Filling the open end of the segment $(\lambda f.\, \lambda x.\, \omega_{f,x})\, g$ by the λ-term $f\, x\, x$ results in the λ-term $(\lambda f.\, \lambda x.\, f\, x\, x)\, g$. In the result, the free variables f and x of the λ-term become bound by the binders λf and λx of the segment, respectively.

Considering the structure of segments and the operations defined on segments, segments may rightfully be called a special kind of λ-contexts. The open end acts

as a hole. Filling the open end by a λ-term or a segment amounts to filling a hole of a λ-context: this comparison regards both the structure of the result (a λ-term or a segment) and the intentional variable capturing. The distinguished properties of segments with respect to λ-contexts are the uniqueness of the open end occurrence and its specific position.

The comparison of segments with λ-contexts entails the same problems with formalisation. Extending λ-calculus with segments involves the same problems regarding implementation of variable capturing and the β-rewrite relation on segments, which have been described in Section 2.2.

De Bruijn solved these problems in a way that is very different from our approach in the framework λc. The difference in the approaches comes primarily from different aims: de Bruijn wanted a calculus which was implementation-oriented, that is easy to implement in Automath, whereas we wanted a calculus which was close to λ-calculus. De Bruijn's choices in designing a segment calculus come from his intended applications of segments. In these applications an abbreviation for segments was desirable, and these abbreviations were used in a very specific way. We illustrate this abbreviation mechanism by an example.

Example 5.1.3 See Figure 5.3. The figure shows two terms in tree notation. In the tree on the left, a segment of a term has been included in a box. In the tree on the right, that segment has been abbreviated as follows:

- a segment variable $t_{f,x}$ (labelled and with arity 1) has been introduced and the segment has been replaced by this variable; the label of t consists of the variables that are bound by the segment in the argument of t, and the label binds the free occurrences of f and x in the argument of t;
- the abstraction λt has been introduced;
- the segment has been extended with the explicit open end $\omega_{f,x}$; and
- the application has been introduced with the segment as an argument.

The application–abstraction pair serves as the definition of the segment $t = (\lambda f.\, \lambda x.\, \omega_{f,x})\, g$ in $t_{f,x}(f\, x\, x)$.

The figure in the example shows how segment variables are introduced. By looking at the figure from left to right, the figure shows how a variable for a segment can be introduced, coupled with the segment and used. By looking at the figure from right to left, the figure shows how a segment variable can be substituted by a segment.

The usage of segment variables is remarkable: whenever a segment variable occurs, it is provided with an object to be put into the open end of the segment for which this variable holds place. Hence, segment variables are never used on their own, without this argument. Moreover, when a segment variable is substituted by a segment, this argument is immediately placed into the open end of the segment.

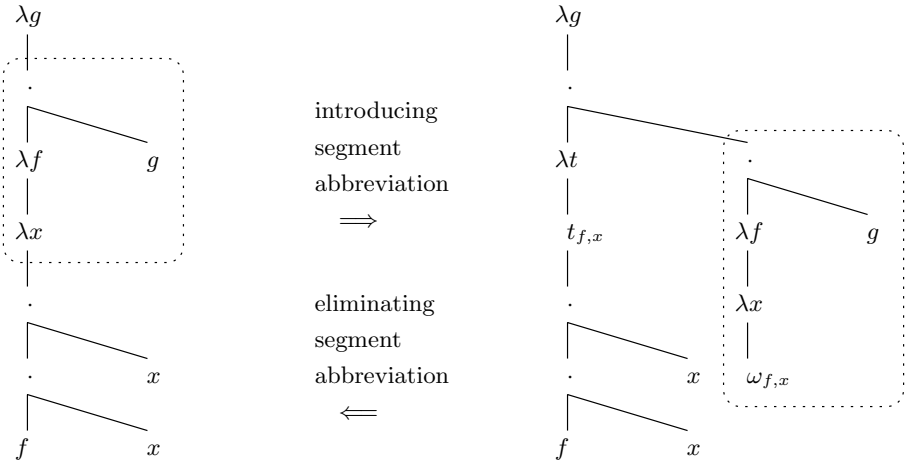


Figure 5.3: Abbreviations for segments

That is, hole filling is immediately computed upon substitution for a segment variable by a segment. Note that when a segment variable is used, it is labelled by the binders of the segment that it holds place for; in the example, the occurrence of the segment variable is $t_{f,x}$, and f and x are (the names of) the binders of the segment. Moreover, the label matches the label L of ω_L of the segment.

Such a manipulation with segments is fine-tuned for applications of segments. We will first describe these applications of segments and then return to some details of de Bruijn's segment calculus.

Application of segments

Segments were introduced to facilitate the use of abbreviations for segments of λ-terms in the family of Automath languages.

Before going into applications of segments, we devote a few words to the Automath project. The Automath project started in 1967 under the auspices of N. G. de Bruijn. The primary goals of the project were the following:

- first, to design a language, in the words of de Bruijn (see [NGdV94] page 73), 'suitable for expressing very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of grammar are obeyed'; and

- second, to design computer programs to automate the verification of Automath texts.

That is, the goals were to design a logical framework and implement it as a proof-checker. Three decennia of research resulted in a family of typed λ-calculus-like

formal languages. We name a few: AUT-68, AUT-QE, AUT-Π, AUT-SL, AUT-SYNT, AUT-ΔΠ, and AUT-QE-NTI. Several of the Automath languages were implemented.

For an introduction to the Automath project the reader is referred to [Bru67, Daa73, Bru80]; for a description of Automath languages to [Daa80, Ben81]; for examples of mathematical theories formalised in Automath to [Ben77, Zuc75].

One of the Automath languages is of particular interest to segments, namely AUT-SL (Single Line, see [Bru71, Ben77]). The reason why it is interesting is the following. In general, Automath texts, called *books*, consist of lines. Each line determines a basis, declares an object or an axiom, or defines an object. An Automath book is correct if it satisfies certain well-formedness requirements, including certain well-typedness requirements. Checking the correctness of an Automath book involves also reasoning about bases, declarations and definitions. Hence, checking the correctness of a book involves some meta-reasoning. In contrast, in AUT-SL Automath books are represented as one large λ-term. Bases, declarations and definitions are encoded as parts of such a λ-term. The advantage of such a representation is that the correctness of an Automath book corresponds to the well-typedness of a λ-term. Accordingly, studying properties of Automath books and transformations defined on them boils down to studying properties of a typed lambda calculus. The development of AUT-SL has led to Nederpelt's Λ (also called AUT-Λ, see [Ned73]), for which the first proof of strong normalisation for an Automath language was given.

However, there is also a disadvantage of such a book representation: there may occur many repetitions of segments. The size of a book representation can be reduced by introducing an abbreviation mechanism (in the sense of Remark 1.2.16) for segments: notation for segments, segment variables and abstractions over segment variables. Hence, a segment *calculus* is desirable.

Back to the applications of segments. In the literature we find two applications of segments in languages such as AUT-SL. We describe the applications and give an example of each. Note that both applications involve a *typed* lambda calculus with segments.

i) Segments can be used for representing abstract mathematical structures such as groups, linear orders, and vector spaces, within lambda calculus. Such structures are sequences of declarations and axioms:

$$x_1 : A_1, \ldots, x_n : A_n.$$

Instances of such structures are also tuples:

$$(a_1, \ldots, a_n) \text{ with}$$
$$a_1 : A_1, a_2 : A_2[\![x_1 := a_1]\!], \ldots a_n : A_n : [\![x_1 := a_1, \ldots, x_{n-1} := a_{n-1}]\!].$$

In fact, for this application only the segments of a special form are needed: the segments consisting only of abstractions and the segments consisting only

of applications. The segments consisting only of abstractions are called *telescopes*. The segments consisting only of applications are called *vectors*. Then an abstract structure can be represented as the telescope

$$S \equiv \lambda x_1 : A_1 \ldots . \lambda x_n : A_n . \omega_{\vec{x}},$$

and an instance of S can be represented as the vector

$$\underline{a} \equiv \omega\, a_1 \ldots a_n.$$

The vector $\underline{a}$ is said to 'fit' into the telescope S.

ii) Segments can be used to uniformly represent an Automath book, line by line. As already said, an Automath book consists of lines and each line is a basis, declaration or a definition. Each of these lines can be represented as a segment. Then translation of a book, line by line, corresponds to combining segments that represent these lines, one by one.

We give an example of each application in turn. In both examples it is necessary to have segments and segment abbreviations at the same level as λ-terms. Only in that case can segments be freely manipulated: they can be denoted; segment variables and abstractions over segment variables can be used in abbreviations; segments can be passed as arguments of a function; and segments may be involved in the standard transformations including substitution and β-rewriting.

In both examples we use types in abstractions over term variables and leave the types of abstractions over segment variables out. We depict both examples as (quasi-)λ-trees. We say 'quasi', because the trees will also include λ-terms as subtrees (due to the lack of space). Moreover, in tree notation, the type of the abstraction over a term variable is denoted together with the term variable.

Example 5.1.4 (Segments for mathematical structures) This example is about a reflexive–euclidic binary relation. A binary relation R over a set S is reflexive if and only if $\forall x \in S.Rxx$, and it is euclidic if and only if $\forall x, y, z \in S$ if Rxy and Rxz then Ryz. This example is also about two properties of such a relation: transitivity and symmetry. One can show that for such a relation R it holds $\forall x, y, z \in S$ if Rxy and Ryz then Rxz, and $\forall x, y \in S$ if Rxy then Ryx.

In a λ-calculus with segments, the structure of a reflexive–euclidic binary relation can be represented by the segment

$$\begin{aligned} \underline{re} \equiv \quad & \lambda S : Set. \\ & \lambda R : (S \to S \to Prop). \\ & \lambda r : (\Pi x : S.Rxx). \\ & \lambda e : (\Pi x : S.\Pi y : S.\Pi z : S.Rxy \to Rxz \to Ryz).\ \omega_{S,R,r,e}. \end{aligned}$$

It can be defined by using an abstraction over a segment variable t and using an application with $\underline{re}$ as the right argument, as in Figure 5.4. The rest of the λ-term in the figure contains the proofs *trans* and *sym* that R is transitive and symmetric, respectively. The reader is invited to check the proof terms.

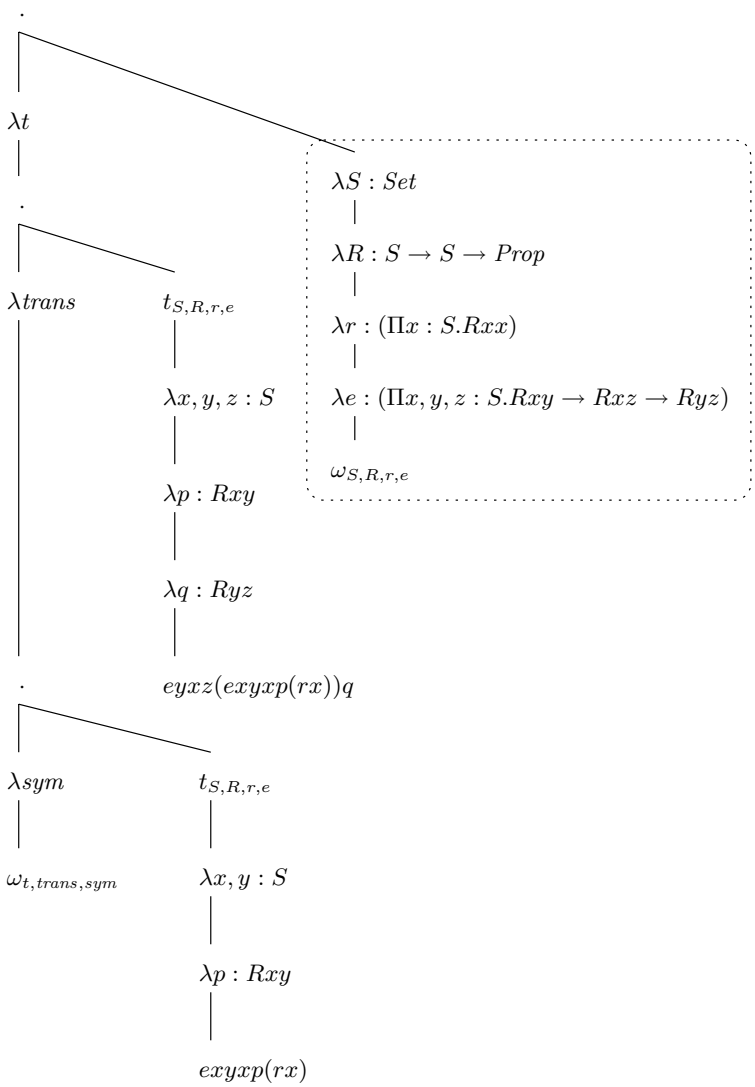


Figure 5.4: A reflexive–euclidic relation is transitive and symmetric

Example 5.1.5 (Segment calculus for Automath) We follow the example given in [Bru78] (p. 35). We first give an example of an Automath book. Then we represent this book as a part of AUT-SL-term, and as a segment.

Before going into the details of the example, we explain the Automath notation that is essential in this example. In Automath there is only one binder, namely $[x : A]$. Hence, there is no distinction between the binders $\lambda x : A$ and $\Pi x : A$ as in for example the λ-cube. For example, the λ-term and its type $(\lambda x : \alpha.\, \lambda y : \beta.\, x) : (\Pi x : \alpha.\, \Pi y : \beta.\, \alpha)$ are denoted in Automath as $([x : \alpha][y : \beta]x) : ([x : \alpha][y : \beta]\alpha)$. Such a notation has an advantage when using segments: for example, if $s \equiv [x : \alpha][y : \beta]$ then s can be used as abbreviation in the term as well as in its type: $s(x) : s(\alpha)$.

Back to the example of an Automath book. We will represent the following Automath book[1]. The lines are numbered for easy reference.

	context indicator	identifier		definition	category
(1)	$-$	α	$:=$	$-$	`type`
(2)	α	x	$:=$	$-$	α
(3)	x	y	$:=$	$-$	α
(4)	y	f	$:=$	PN	α
(5)	x	b	$:=$	$f\alpha xx$	α

We explain the book line by line.

i) The first line contains a definition of the basis $\alpha : \texttt{type}$.

ii) The second line contains a definition of the basis $\alpha : \texttt{type}, x : \alpha$.

iii) The third line contains a definition of the basis $\alpha : \texttt{type}, x : \alpha, y : \alpha$.

iv) The fourth line is a declaration of f of type $[\alpha : \texttt{type}][x : \alpha][y : \alpha]\alpha$. Here, PN stands for Primitive Notion and it is predefined.

v) The last line is a definition of b as $[\alpha : \texttt{type}][x : \alpha]f\alpha xx$ of type $[\alpha : \texttt{type}][x : \alpha]\alpha$.

The main part of this book is the declaration of f and the definition of b, whereas the first three lines are a stepwise formation of the basis used in the declaration and definition. Hence, in the Automath language AUT-SL this book is represented[2] as follows:

$$[f : [\alpha : \texttt{type}][x : \alpha][y : \alpha]\alpha]\ ([b : [\alpha : \texttt{type}][x : \alpha]\alpha]\ \ldots)\ ([\alpha : \texttt{type}][x : \alpha]f\alpha xx).$$

[1] This book is written in the Primitive Automath Language (PAL), which has no abstractions.

[2] Actually, in the Automath book, the parameter mechanism has been used in the declaration of f and the definition of b. By the parameter mechanism, we mean that the function f is defined as $f(\alpha, x, y) = t$, as opposed to the definition of f using the λ-abstraction of lambda calculus, as in $f = \lambda\alpha.\, \lambda x.\, \lambda y.\, t$. The parameter mechanism is weaker than the λ-abstraction. Thus, the representation of the Automath book given here is an embedding into a stronger system. However, this does not weaken the argument of the example.

Note that this is an unfinished AUT-SL-term, where the dots ... denote the missing part. As we mentioned earlier, AUT-SL does not support segments, so this is an unfinished AUT-SL-term.

See now Figure 5.5. The figure shows a segment. The dashed lines separate the representations of the Automath lines; the parts are numbered accordingly. We explain the segment part by part.

i) The first part contains a definition of the basis $\alpha : \texttt{type}$. The definition is represented by the segment $[\alpha : \texttt{type}]\omega_\alpha$, using the segment variable s and an abstraction–application pair.

ii) The second part contains a definition of the basis $\alpha : \texttt{type}, x : \alpha$. The definition is represented by extending the existing context s, in an analogous way as in the first part.

iii) The third part contains a definition of the basis $\alpha : \texttt{type}, x : \alpha, y : \alpha$. The definition is represented by extending the existing context t, in an analogous way as in the first and second part.

iv) The fourth part is a declaration of f of type $[\alpha : \texttt{type}][x : \alpha][y : \alpha]\alpha$. The type of f is given by using the abbreviation for the segment u and filling its open end by α.

v) The last part is a definition of b as $[\alpha : \texttt{type}][x : \alpha]f\alpha xx$ of type $[\alpha : \texttt{type}][x : \alpha]\alpha$. Both the definition and type of b use the abbreviation for the segment u.

In this representation of the Automath book given above, abbreviations (i.e. segment variables) have been used for parts of the term that occur repeatedly. Though, there is some overhead in the segment due to the formation of the bases. However, this overhead is not bad. In general, in a (bigger) Automath book, the same bases are often used many times, so it pays off to define the bases once as segments and use the segment variables (abbreviations!) in the remainder of the book.

Historical remarks

The origin and development of telescopes, segments and segment calculus are related to the Automath project and the people working on that project. In the early formalisations of pieces of mathematics, de Bruijn proposed using telescopes for representing abstract structures. In [Zuc75], where a piece of classical real analysis has been formalised in the Automath language AUT-Π (telescopes where not included in AUT-Π), J. Zucker mentions that telescopes would 'especially be useful in an Automath treatment of abstract algebra.' In [Ben77], where Landau's 'Grundlagen' have been translated into Automath, L.S. van Benthem Jutting also mentions that telescopes are useful structures. In the Automath language AUT-SYNT, proposed and partly developed by I. Zandleven, telescopes were incorporated; however AUT-SYNT was never implemented.

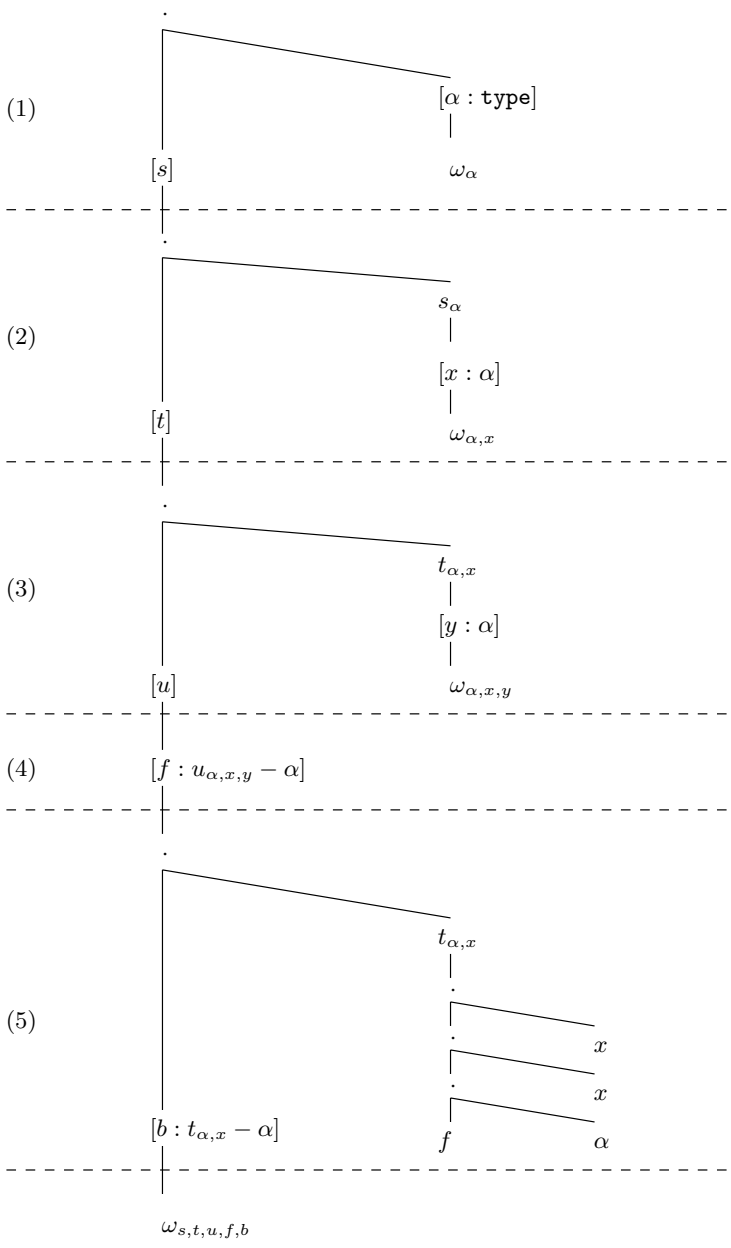


Figure 5.5: Example of an Automath book

In [Bru78] de Bruijn defined the segment calculus, which we study in this chapter. In that report, de Bruijn motivates the use of segments in a broader sense as described by our second motivation, as a logical framework. De Bruijn contributed the weak confluence proof to R. Wieringa and L.S. van Benthem Jutting, who checked this independently. This report contains the untyped and a simply typed version of the calculus where types are generated over one type constant.

In [Bal86, Bal87], H. Balsters defined the calculus $\lambda\sigma$, which is a restriction of the original calculus. H. Balsters proved that $\lambda\sigma$ has the confluence property and that its simply typed version $\lambda_T\sigma$ (where types are generated over one type constant) has the subject reduction property.

In [Bru91] de Bruijn described telescopes in a general setting with dependent types. This is an abstract approach, independent of a particular lambda calculus. It describes a calculus on telescopes, including composition (concatenation) of telescopes and tuples that fit into telescopes. Telescopes were not typed in this system. The focus is on telescopic mappings, which represent morphisms between the abstract structures that telescopes represent.

Details of de Bruijn's segment calculus

The original calculus is called $C\lambda\xi\phi\omega\eta$. It is an extension of a name-free untyped lambda calculus. Among the added features are: a symbol for denoting a hole, segment variables, functions over segments and terms, and renamings for adjusting indices. With segment variables and functions over segments, and with β-reduction defined on both terms and segments, this calculus, as far as the author of this thesis knows, is the first context calculus.

We go here into some details of the calculus $C\lambda\xi\phi\omega\eta$. We start with a description of the terms of $C\lambda\xi\phi\omega\eta$, continue with rewriting on these terms and typing of these terms, and end with an overview of the properties of $C\lambda\xi\phi\omega\eta$.

Signature and formulas. Terms of the calculus $C\lambda\xi\phi\omega\eta$ are called *formulas* by de Bruijn. The formulas of the calculus are divided into two sorts: terms, with a variable $\xi(n)$ at the end of the spine; and segments, with an open end $\omega(k,\theta)$ at the end of the spine.

The formulas are built from a signature. The signature is an extension of the signature of lambda calculus with function symbols of fixed arity. The extensions pertain to the notation for segments, the notation for segment variables and the notation supporting transformations on segments and terms.

We give the definition of the formulas of $C\lambda\xi\phi\omega\eta$. We deviate here from de Bruijn's notation using 'combs', and use a more usual notation (cf. also [Bal87]). The formulas of $C\lambda\xi\phi\omega\eta$ are defined by

$$t ::= \xi(n) \mid \lambda t \mid \delta t\, t \mid f(\vec{t}) \mid \phi(\theta)t \mid \omega(k,\theta) \mid \eta(k,n)t$$

and, as can be seen from this formulation, the signature consists of the following symbols: (in the signature, ξ, λ, δ, ϕ, ω and η are fixed (parts of the) symbols)

- (term) variables: denoted by $\xi(n)$ where $n \in \mathbb{N}$ and $n \geq 1$. The number n pinpoints the binder by which this variable is bound, as it is customary for name-free variables. That is, n is a 'de Bruijn's index'.

- abstractor: unary operator denoted by λ. This abstractor is used for binding both term and segment variables.

- applicator: binary operator denoted by δ. Applicator is, contrary to the lambda calculus, in this calculus explicitly written. The order of arguments of an applicator is reversed, viz. $\delta a f$, contrary to the standard notation $f a$. This inverse notation has the advantage that in a β-redex $\delta a(\lambda x.M)$, the argument a is easy to see being between δ and λ.

- function symbols: with fixed arity.

- reference transforming mappings: unary operators denoted by $\phi(\theta)$ where θ is a mapping with $\theta : \{1, \ldots\} \to \{1, \ldots\}$. The mapping θ adjusts the indices, which is necessary during β-rewriting. In this calculus β-rewriting is defined stepwise, where the application–abstraction pair $(\lambda x.\_)t$ traverses through an object, thus permuting the order of abstractors.

- the open end or end–of–segment symbol: nullary operator denoted by $\omega(k, \theta)$ where $k \in \mathbb{N}$ and θ is a mapping with $\theta : \{1, ..., k\} \to \{1, \ldots\}$. This symbol lies at the end of the spine of a segment. The natural number k is the number of abstractors at the spine of the segment that can bind free variables in an object placed into $\omega(k, \theta)$. The mapping θ keeps track of permutations of abstractions which occur during rewriting.

- segment variables: unary operators denoted by $\eta(k, n)$ where $k, n \in \mathbb{N}$ and $n \geq 1$. The first number k denotes the number of abstractors lying on the spine of the segment that this variable holds place for. The second number n denotes by which binder this variable is bound. That is, n is a 'de Bruijn's index'.

The integer k in $\omega(k, \theta)$ is called the weight of the segment where $\omega(k, \theta)$ lies on the spine.

The notation is rather straightforward. The variables, abstractor and applicator are the same as in lambda calculus in name-free notation. The reference transforming mappings are introduced only for an efficient performance of transformations on these elements. The end–of–segment symbol denotes, as the name suggests, the end of a segment. The only symbol that needs some explanation is the segment variable. Segment variables are unary. Abbreviations of segments are used only to form something else, a new segment or a term; they are not used on their own.

Example 5.1.6 We give some examples of the formulas of $C\lambda\xi\phi\omega\eta$. Let id_k denote the identity function from $\{1, \ldots, k\}$ to $\mathbb{N}$.

i) The segment $(\lambda f.\, \lambda x.\, \omega_{f,x})g$ of Example 5.1.3 (see Figure 5.3) is represented as

$$\delta\xi(1)\lambda\lambda\omega(2, id_2).$$

ii) The whole element with the segment of the same example (Example 5.1.3, see the element on the right of Figure 5.3) is represented as

$$\lambda\delta(\delta\xi(1)\lambda\lambda\omega(2, id_2))\lambda\eta(2, 1)(\delta\xi(1)\delta\xi(1)\xi(2)).$$

iii) The representation of the context $C \equiv (\lambda y.\, [])x$, which is a segment, of Example 2.5.3 is

$$\delta\xi(1)\lambda\omega(1, id_1).$$

iv) An example of a formula of $C\lambda\xi\phi\omega\eta$ is also

$$\phi(\theta_1)\delta\xi(5)\lambda\lambda\omega(3, \theta_3)$$

with $\theta_1(5) = 4$ and otherwise $\theta_1(n) = n$; and, $\theta_2(1) = 2$, $\theta_2(2) = 1$ and $\theta_2(3) = 3$.

Transformations on formulas. The transformations on formulas that we are interested in are substitution for term and segment variables, β-rewriting, hole filling and composition.

Remark 5.1.7 An important issue in a lambda calculus with segments in general, is that the specific structure of segments regarding the position of the hole is preserved under transformations. The preservation of the segment structure means in particular that if a segment is involved in a transformation, the hole on the spine can be neither multiplied nor erased *within* (the boundaries of) the segment. That is, descendants of a segment are segments again.

The preservation of the segment structure under substitution, hole filling, composition and α-reduction, if applicable, can easily be verified. The preservation under β-reduction will now be explained in some detail. Consider a segment C and a β-redex $(\lambda x.\, s)t$ and distinguish the relative position of C with respect to the redex in a β-step. If the segment C is subsumed by the redex, then the segment can only be manipulated as a whole (in a structure preserving way). That is, by a β-step C may be multiplied, but the hole of C can neither be multiplied nor can it change the position *within* C. If the segment C subsumes the redex, then the hole is either at the spine of the redex or disjoint from the redex. In all these cases the hole remains on the spine under the contraction of the redex. This is due to the specific position of the hole in a segment and it is in general not the case with contexts with one hole at an arbitrary position: for example, if $(\lambda x.\, yxx)\,[]$ is a λ-context, then it reduces to the λ-context $y\,[]\,[]$, which has two holes.

In $C\lambda\xi\phi\omega\eta$, substitution for term or segment variables is *not* defined as a meta-operation, because there is no need for meta-substitutions. In $C\lambda\xi\phi\omega\eta$, substitution is implemented together with β-rewriting by the rewrite relation called *single step reduction* $>_1$. This reduction rewrites β-redexes stepwise, as it has become a tradition in explicit substitution calculi (see [ACCL91]), where, in name-carrying lambda calculus notation, the pair $(\lambda x.\_)t$ traverses through the term or segment s in the redex $(\lambda x.\, s)t$. In $C\lambda\xi\phi\omega\eta$, the single step reduction is defined by induction to the structure of s in the redex $\delta t\lambda s$. During the traversal of the pair $\delta t\lambda\_$, new reference transforming mappings are incorporated in the formula in order to adjust the references. The adjustment of references is necessary because of the permutation of binders that occurs during the traversal.

The single step reduction consists of two sets of rules: nine rules (cf. [Bru78], rules (A1) through (A9)) for traversing reference transforming mappings $\phi(\theta)$ in the redex $\phi(\theta)s$, and eleven rules (cf. [Bru78], rules (B1) through (B11)) for traversing the pair $\delta t\lambda\_$ in the redex $\delta t\lambda s$. The rules are technical but straightforward. We mention a few cases. Consider a redex $\delta t\lambda s$.

- Suppose s is the open end $\omega(k,\theta)$ and according to θ there is a reference to this λ. Then the traversing pauses at the open end until eventually the open end is filled. This is not a particular rewrite rule, but it follows from a couple of relevant rules.

- s is a term variable $\xi(1)$ which refers to this λ: if t is a term, the variable is replaced by t, and the pair $\delta t\lambda\_$ is dropped. This is the rule $(B3)$.

- Suppose s is an abstraction $\lambda s'$. Then the pair $\delta t\lambda\_$ passes over the λ of s, the $\delta t\lambda\_$ is applied to s' and some reference numbers are adapted in both s' and t because the two λ's are permuted. The adaptation is done by a reference transforming mapping, so the redex $\delta t\lambda\lambda s'$ reduces to $\lambda\delta\phi(\theta_1)t\lambda\phi(\theta_2)s'$ for certain θ_1 and θ_2. This is the rule $(B7)$.

- Suppose s is $\eta(k,n)s'$ where the segment variable $\eta(k,n)$ does not refer to the abstractor of the redex (i.e. $n > 1$). Then the pair $\delta t\lambda\_$ passes over η, the reference number of η is decreased by 1 (one less abstraction on the left of it), and the pair $\delta t\lambda\_$ is applied to the argument of the segment variable with adapted reference numbers, because the binders of η and the λ are permuted. This is the rule $(B8)$.

- Suppose s is $\eta(k,1)s'$, t is a segment without reference transforming mappings on the spine, and $t \equiv t'\omega(k,\theta)$. Then the variable η is replaced by t' (the segment without the open end), the pair $\delta t\lambda\_$ is passed to s' with some adaptation of reference numbers, and θ is applied to s' and t of the pair to establish the correct binding between t' and the rest. This is the rule $(B9)$.

With β-rewriting defined via the single step reduction, the calculus $C\lambda\xi\phi\omega\eta$ is a lambda calculus with explicit substitutions. In fact, the calculus $C\lambda\xi\phi\omega\eta$ is, as far as the author of this thesis knows, one of the first explicit substitution calculi.

There are two main differences with the modern calculi with explicit substitutions. First, the single step reduction in $C\lambda\xi\phi\omega\eta$ is defined on more expressions, because $C\lambda\xi\phi\omega\eta$ includes segments. Second, in a lambda calculus with explicit substitutions, rewriting of a β-redex $(\lambda x.\, s)t$ is set off by explicitly forming a substitution $[\![x := t]\!]$ and applying it to s, viz. $(\lambda x.\, s)t \to s[\![x := t]\!]$. In $C\lambda\xi\phi\omega\eta$ the pair $(\lambda x.\, \_)t$ traverses through s as an application–abstraction pair.

For the rules to work properly, some conditions are imposed: one of the conditions is imposed on the form of the segments, and the other conditions are incorporated in the rewrite relation.

The condition on the form of segments is called the internal reference condition, and it is the following: the mapping θ in $\omega(k, \theta)$ of a segment S ranges over the binders present in S (possibly hidden within a segment variable on the spine of S); the mapping does not range over binders outside the segment S. This condition is imposed because otherwise rewrite steps may lead to inconsistent bindings (an example is given in [Bru78], page 28). However, if a segment satisfies the internal reference condition, then all its reducts satisfy the internal reference condition (that is, the internal reference condition is preserved under rewriting).

The conditions in the rewrite relation are related to ensuring proper usage of variables and they are the following. Consider the redex $\delta t\lambda s$.

- If t is a term, then all variables of s that refer to this λ are term variables, and if t is a segment, then all variables of s that refer to this λ are segment variables.

- If this formula is of the form $\delta t\lambda\eta(k, 1)s'$ and if t is a segment, then the weight of t equals k and t may not contain reference transforming mappings on the spine. De Bruijn proved that rewriting with respect to the rules $(A1)$ through $(A9)$ (involving computation of reference transforming mappings) are strongly normalising, so the condition that t may not contain reference transforming mappings on the spine can always be fulfilled; the condition only imposes an order on rewriting.

These conditions on the rewrite relation call for typing.

Hole filling and composition are only implicitly present in the calculus. Both hole filling and composition are denoted by application of a segment variable to an object. For example, in $\eta(3, 1)\delta a\xi(5)$ and $\eta(2, 1)\lambda\omega(3, \theta)$. However, upon substitution of a segment s for the segment variable $\eta(k, n)$ in $\eta(k, n)u$, the open end of s is immediately filled by u. So a segment may be abbreviated by a variable, but substitution of the variable and hole filling or composition are done in one rewrite step. Such an approach is close to how segments are used in proof-checking.

As a whole, the original calculus reflects the very 'Automath-ed', implementation-oriented fashion of the object processing: the objects are represented as strings of characters and the transformations are implemented by cutting the strings,

duplicating parts of the strings, inserting new characters and gluing these parts together again.

Typing. The principal role of types in the calculus is to avoid deadlocks. According to de Bruijn, deadlock occurs in three situations:

- when a function over a term variable is applied to a segment,
- when a function over a segment variable is applied to a term, or to a segment with a different weight than expected (by that segment variable in the function body) and
- when a segment does not satisfy the internal reference condition.

These situations are precisely the conditions that are imposed by the rewrite rules, and which we discussed above. As already said above, the last situation is avoided by considering only the set of formulas which satisfy the internal reference condition, which is closed under rewriting. The first two situations are avoided by typing.

Types are called *frames* by de Bruijn. Frames and frame functions are introduced in $C\lambda\xi\phi\omega\eta$ as a finite decision procedure for avoiding deadlocks of the first two kind.

Frames are defined over one type constant. This implies in particular, that frames of λ-terms in $C\lambda\xi\phi\omega\eta$ are basically simple types over a singleton.

Frames of segments are rather complex. Due to the distinguished position of the open end in a segment, the frame of a segment is 'open-ended'. Due to this 'open-endedness', frames of segments can be considered to be polymorphic. We explain this in an informal way, in lambda calculus using meta-contexts, for which the same argument holds.

Intermezzo 5.1.8 We describe the form of the type of a segment and argue that it is, in a sense, polymorphic.

Consider simply typed lambda calculus where types are generated over the singleton $\{\mathtt{t}\}$. Let C be a meta-context with the hole at the end of the spine. In general, the form of the meta-context C restricts the type of terms t that may be placed into the hole, but the type of $C[t]$ eventually depends on the type of t. Take, for example, the meta-context $C_1 \equiv \lambda x : \mathtt{t}.\,[]\,x\,x$. If a λ-term t_1 is to be placed into the hole, its type should be an arrow type $\mathtt{t} \to \mathtt{t} \to T$, where T denotes an arbitrary type, and the type of the result $C_1[t_1]$ is then $\mathtt{t} \to T$.

In general, in C there may also be some applications and abstractions on the spine. These applications and abstractions can be matched, after some rewrite steps (on the representation of the meta-context in a context calculus), to form a redex $(\lambda x.\,s)\,s'$. Here, the type of the argument s' cancels the type of the first argument of the function $\lambda x.\,s$, so the type of $(\lambda x.\,s)\,s'$ is the type of s. By continuing the rewriting to match all possible redexes[3] one obtains a context with unmatched

[3]This suggests at least weak normalisation of this rewriting process. Normalisation properties have not been proved for de Bruijn's calculus. Our segment calculus λc^s (Section 5.2), which is a simply typed lambda calculus with segments, has the strong normalisation property (Theorem 5.2.20).

abstractions $\lambda\vec{x}$ at the front, and with unmatched applications $\vec{s}$ in front of the hole $[]\,\vec{s}$, thus:

$$\lambda\vec{x}.\,[]\,\vec{s}.$$

Then, the type of the meta-context is determined by the types of unmatched abstractions in front, and the types of unmatched applications at the hole. Accordingly, if a λ-term t is to be placed into the hole, its type should be $\vec{\tau} \to T$, where $\vec{\tau}$ denote the types of unmatched applications at the hole and T denotes an arbitrary type. The type of a result $C[t]$ is then $\vec{\sigma} \to T$, where $\vec{\sigma}$ denote the types of unmatched abstractions. For example, in the meta-context $C_2 \equiv \lambda x.\,\lambda y.\,(\lambda z.\,\lambda u.\,[]\,d)\,b\,c$ the abstractors λx and λy and the applicator $\_\cdot d$ remain unmatched. If a λ-term t_2 is to be placed into the hole, its type should be $\mathtt{t} \to T$, where T denotes an arbitrary type, and the type of the result $C_2[t_2]$ is then $\mathtt{t} \to \mathtt{t} \to T$.

Hence, the type of (the representation in a context calculus of) a meta-context can be described by these two sequences of types: the types of unmatched abstractors and the types of unmatched applicators. In addition, the types of binders in whose scope the hole is define the type of the communication between the context and a term placed into the hole. (The communication and its type have been ignored in the argument above.)

In $C\lambda\xi\phi\omega\eta$, the situation resembles the situation in lambda calculus with contexts as described in Intermezzo 5.1.8. The only difference is that in $C\lambda\xi\phi\omega\eta$ there are two kinds of abstractors: when matching abstractors to applicators we have to take into account both λ's and η's. In $C\lambda\xi\phi\omega\eta$, the frame of a segment S consists of three sequences of types:

- the sequence of types of binders that remain in front of the segment after some rewrite steps;
- the sequence of types of applicators that remain in front of the open end after some rewrite steps; and
- the sequence of types of the binders in whose scope the open end is in S.

Example 5.1.9 We consider the meta-context C_2 of Intermezzo 5.1.8 and write it in the notation of $C\lambda\xi\phi\omega\eta$. So, let $S \equiv \lambda\lambda\delta\xi(6)\delta\xi(5)\lambda\lambda\delta\xi(7)\omega(4, id)$ where id denotes the identity function. The frame of S is $(\mathtt{t},\mathtt{t};\ \mathtt{t};\ \mathtt{t},\mathtt{t},\mathtt{t},\mathtt{t})$, where the three sequences of types are as described above. Here, we deviate from the notation for frames as in [Bru78].

Frames are computed by a frame function. The frame function takes an element as an argument and computes its types, if it exists. Next to typing λ-terms and segments of $C\lambda\xi\phi\omega\eta$ as described above, the frame function defines under which conditions formulas can be combined into bigger, also well-typed formulas. Typing defined via a (frame) function is a different style of typing than in for example PTSs. The main difference is in the use of bases, which are present in PTSs, and

which are not present in a frame function. However, basically these presentations are the same.

Results. We sum up the results proved for $C\lambda\xi\phi\omega\eta$ or its variations. De Bruijn proved that the rewriting with respect to the rules $(A1)$ through $(A9)$ is strongly normalising. R.M.A. Wieringa and L.S. van Benthem Jutting both proved independently that $C\lambda\xi\phi\omega\eta$ is weakly confluent. H. Balsters defined the calculus $\lambda\sigma$, a restriction of $C\lambda\xi\phi\omega\eta$ which does not involve reference transforming mappings. H. Balsters proved that $\lambda\sigma$ is confluent. He also defined a framed version $\lambda_T\sigma$ and proved that it has the subject reduction property.

5.2 Segments in the context calculus λc

The calculus λc^s, defined in this section, can be described in one sentence: it is the context calculus λc with types for the simply typed lambda calculus with polymorphic segments. We explain this description into more detail; see also Figure 5.6.

- The calculus λc^s is the context calculus λc equipped with types: Like the calculi defined in Chapter 4, the calculus λc^s is a subsystem (in the sense of Definition 1.1.14) of the framework λc, defined by a type system.

- Segments in the framework λc: As explained in Section 5.1, a segment is a special kind of contexts, with a unique hole occurrence placed at the end of the spine. Being a context, each segment can be represented in the framework λc, where the typing rules take care of the restrictions.

- Segments in the calculus λc^s: By the term formation rules of λc^s, segments can be represented in λc^s. In addition, the calculus λc^s includes segment variables and functions over segments. That is, the calculus λc^s is a calculus with segments as first-class objects.

  We emphasise that λc^s incorporates the notion of segment, but in a different way than de Bruijn's segment calculus. We will return to this comparison later in the section.

- The calculus λc^s is a simply typed lambda calculus with segments: The type system by which the calculus λc^s is defined, is basically the simply typed lambda calculus. That is, in λc^s the terms and segments of the simply typed lambda calculus can be represented.

- Polymorphic segments: Due to the special position of the hole in a segment, the type of the hole (and hence, the type of the segment too) is not fully determined (see Intermezzo 5.1.8). The segments are typed in a way that this polymorphism is maintained.

This section is structured as follows. First we will briefly discuss the representation of segments within the framework λc. Next, we will define the calculus λc^s and show that λc^s is a subsystem of the framework λc. Then we will show that

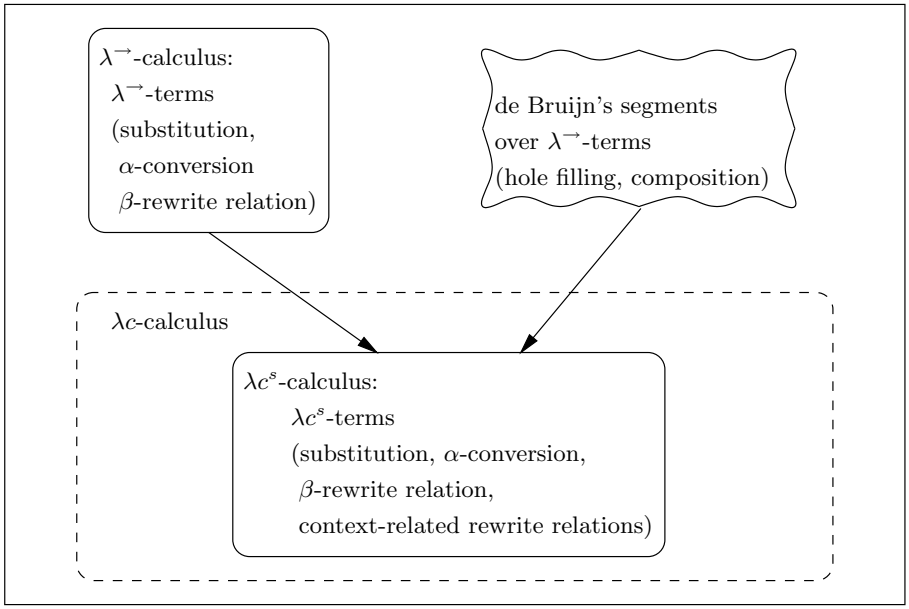


Figure 5.6: The calculus λc^s

λc^s is confluent and strongly normalising. We will compare the calculus λc^s to de Bruijn's segment calculus, which also implements the notion of segment, and to the calculus $\lambda c^{\rightarrow}$, which also implements a simply typed lambda calculus with contexts. Finally, we will give some (non-)examples of typing in λc^s.

The approach to the formalisation of segments in λc^s for the most part agrees with the approach to the formalisation of λ-contexts in the calculi of Chapter 4. However, the calculus λc^s, and in particular its type system, is more complicated than calculi of Chapter 4 because λc^s deals with the context that have the hole on a designated position and that have a polymorphic type.

Representation of segments in λc

With segments being a special kind of contexts, the representation of the lambda calculus with segments mainly follows the line described in Section 2.5. In particular, segments are represented as abstractions over precisely one hole variable, which occurs at the end of the spine[4]. For instance, the segment $(\lambda x.\,[])\,y$ is represented as $\delta h.\,(\lambda x.\,h\,\langle x\rangle)\,y$. In λc^s, the type system takes care of the position of the hole variable, and the preservation of the hole position under rewriting is guaranteed by the properties of the specific position of the hole variable in a term (see Remark 5.1.7).

[4]The definition of tree representation of λc-terms is a naïve extension of the tree representation of λ-terms, where δ is treated as λ, hole filling as application, Λ as a flattened version of a sequence of λ's and $\langle\,\rangle$ and $\circ$ as a flattened version of a sequence of applications.

The calculus λc^s

We now give the types and the type system of λc^s. The type system, in addition to governing types, guards the polymorphism of segments and the position of the hole. The position of the hole turns up to be easy to monitor, while guarding the polymorphism is rather complex.

The types of λc^s include the types of simply typed lambda calculus and add types for holes, segments and communicating terms. In general, the types of λc^s resemble the types of $\lambda c^{\rightarrow}$ (see Section 4.2), but, in the case of λc^s, the added types are more elaborate because of the polymorphism.

We explain the key issue of types into more detail by recalling Intermezzo 5.1.8. In the remark the segment $C_1 \equiv \lambda x^{\mathtt{t}}.[]\,x\,x$ and its type were considered. It was argued that the type of the term M placed into the hole must be of the form $\mathtt{t} \to \mathtt{t} \to T$, where T denotes an arbitrary type. If such a term M is placed into the hole of C_1, then the type of the result $C_1[M]$ is $\mathtt{t} \to T$. In the calculus λc^s we use type variables α, β and type quantifiers $\forall\alpha$ to implement this kind of polymorphism. A type variable mimics the 'open end' in a type, just as a hole variable stands for the 'open end' in a segment. For example, the segment C_1 is represented by the λc-term $U \equiv \delta h.\lambda x^{\mathtt{t}}.(h\langle x\rangle)\,x\,x$, where the type of the hole is omitted for the time being. The type of the hole $h\langle x\rangle$ is $\mathtt{t} \to \mathtt{t} \to \alpha$, that is, the type of the hole variable h alone is $[\mathtt{t}]\mathtt{t} \to \mathtt{t} \to \alpha$, with the type of communication $[\mathtt{t}]$ added. With such type, both V_1 of type $[\mathtt{t}]\mathtt{t} \to \mathtt{t} \to \mathtt{t}$ and V_2 of type $[\mathtt{t}]\mathtt{t} \to \mathtt{t} \to \mathtt{t} \to \mathtt{t}$ can be put into the hole using the substitutions $[\![\alpha := \mathtt{t}]\!]$ and $[\![\alpha := \mathtt{t} \to \mathtt{t}]\!]$, respectively. By filling the hole with V_1, one obtains a term of type $\mathtt{t} \to \mathtt{t}$, which equals $(\mathtt{t} \to \alpha)[\![\alpha := \mathtt{t}]\!]$. By filling the hole with V_2, one obtains a term of type $\mathtt{t} \to \mathtt{t} \to \mathtt{t}$, which equals $(\mathtt{t} \to \alpha)[\![\alpha := \mathtt{t} \to \mathtt{t}]\!]$. In the tradition of other type systems, the term U is a function of type $[\tau]\tau \to \tau \to \alpha \Rightarrow \tau \to \alpha$ for all α. In the calculus λc^s, this quantification is internalised by $\forall\alpha$ and the type of U is represented by $\forall\alpha.[\tau]\tau \to \tau \to \alpha \Rightarrow \tau \to \alpha$.

We split the set of base types into the set of type variables and the set of type constants.

Definition 5.2.1 (Types of λc^s) Let $\mathcal{V}_b$ denote the set of type variables with α, β as typical elements. Let $\mathcal{V}_c$ denote the set of type constants with $\mathtt{a} \in \mathcal{V}_c$. Let $\mathcal{V} = \mathcal{V}_b \cup \mathcal{V}_c$. Then, the τ-types ($\tau \in \mathcal{T}_s$) and the ρ-types ($\rho \in \mathcal{P}_s$) are defined as

$$\begin{array}{lcl} \tau & ::= & \mathtt{a} \mid \tau \to \tau \mid \forall\alpha.[\vec{\tau}]\vec{\tau} \to \alpha \Rightarrow \vec{\tau} \to \alpha \\ \rho & ::= & \tau \mid \vec{\tau} \to \alpha \mid [\vec{\tau}]\tau \mid [\vec{\tau}]\vec{\tau} \to \alpha \end{array}$$

where $\vec{\tau} \to \alpha$ abbreviates $\tau_1 \to (\ldots(\tau_n \to \alpha))$, and $\forall\alpha$ is a binder which binds both occurrences of α in $\forall\alpha.[\vec{\tau}]\vec{\tau} \to \alpha \Rightarrow \vec{\tau} \to \alpha$. As usual, $\to$ associates to the right, $\to$ binds stronger than $[\,]$ and $[\,]$ binds stronger than $\Rightarrow$.

The free and bound occurrences of type variables are defined in a standard way. In addition, we assume renaming of and substitution for free type variables in $\vec{\tau} \to \alpha$ or $[\vec{\tau}_1]\vec{\tau}_2 \to \alpha$ are defined. Of course, upon substitution, the result has to be a type again. Due to the position of the type variables, only two kinds of types are allowed in a substitution: τ and $\vec{\tau} \to \alpha$.

The intuition behind the types is basically the same as in other applications of the framework λc. The τ-types are used to type λ-terms and segment representations, and the ρ-types are also used for typing holes, communicating objects and subterms of segment representations.

Notation. In the sequel the following notation will be used: $\sigma, \tau, \upsilon, \tau', \vec{\tau}, \vec{\tau}_1 \ldots \in \mathcal{T}_s$, $\rho \in \mathcal{P}_s$.

The rules of the type system are given next. As in the case of the type systems in Chapter 4, in the typing rules there are two bases involved, namely Γ for variables of τ-pretypes and Σ for hole variables. The type of hole variables is of the form $h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha)$. However, in λc^s the base Σ is restricted: it may contain *at most one* declaration over hole variables. We assume without loss of generality that the bases contain distinct variables.

Definition 5.2.2 (Type system) A term $U \in \textsc{Ter}(\lambda c)$ is typable by ρ from the bases Γ, Σ, if $\Gamma; \Sigma \vdash U : \rho$ can be derived using the typing rules displayed in Figure 5.7.

We comment on the typing rules. The rules (*var*), (*abs*) and (*app*) are the rules comparable to the rules (*var*), (*abs*) and (*app*) respectively in the simply typed lambda calculus. In this typing the variables and functions may also range over segments. We will explain the side conditions shortly. The rules (*hvar*) and (*habs*) pertain to typing hole variables and abstractions over hole variables. The rules (*mabs*) and (*mapp*) are used for typing communication.

The rules that need more attention are the rules (*fill*), ($comp_1$) and ($comp_2$). The rule (*fill*) is used for typing hole filling. In this rule, a substitution of the type variable is employed. We give an example of the usage of this rule.

example of (*fill*)
$$\frac{\Gamma \vdash U : \forall\alpha.[\mathtt{a}]\mathtt{a} \to \mathtt{a} \to \alpha \Rightarrow \mathtt{a} \to \alpha \quad \Gamma \vdash V : [\mathtt{a}]\mathtt{a} \to \mathtt{a} \to \mathtt{b}}{\Gamma \vdash U\lceil V\rceil : \mathtt{a} \to \mathtt{b}.}$$

The segment U is of type $\forall\alpha.\rho_1 \Rightarrow \rho_2$ where ρ_1 and ρ_2 end in α. The type ρ_1 is the type of the hole in U. The term V, which is to be put into the hole of U, has a more specific type than the hole: it has the type $\rho_1[\![\alpha := \mathtt{b}]\!]$. Thus, because the segment U is of type $\rho_1 \Rightarrow \rho_2$ for any α, and because V is of type ρ_1 with $\alpha := \mathtt{b}$, the type of $U\lceil V\rceil$ is ρ_2 with $\alpha := \mathtt{b}$, that is, $U\lceil V\rceil : \rho_2[\![\alpha := \mathtt{b}]\!]$.

The two rules ($comp_1$) and ($comp_2$) are used for typing composition. Composition of segments is even more complex than composition of contexts in Chapter 4 because of the polymorphism in types. We explain typing of a composition of segments in λc^s by studying a composition of segments in lambda calculus.

Intermezzo 5.2.3 Here we drop the types at abstractions for the sake of readability.

Note first that in general the type of a redex $(\lambda y'.\, s'')\, s'$ is the type of $s''[\![y' := s']\!]$, which is of the same type as s'' because substitution preserves types. Thus the

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma \vdash x:\tau}$$

$$(abs)\quad \frac{\Gamma, x:\tau;\Sigma \vdash U:\rho}{\Gamma;\Sigma \vdash (\lambda x^{\tau}.U):\tau\to\rho}$$
with $\Sigma=\emptyset$ and $\rho=\tau'$ or $\Sigma\neq\emptyset$ and $\rho=\vec{\tau}\to\alpha$

$$(app)\quad \frac{\Gamma;\Sigma \vdash U:\tau\to\rho \quad \Gamma \vdash V:\tau}{\Gamma;\Sigma \vdash UV:\rho}$$
with $\Sigma=\emptyset$ and $\rho=\tau'$ or $\Sigma\neq\emptyset$ and $\rho=\vec{\tau}\to\alpha$

$$(hvar)\quad \frac{\{h:[\vec{\tau}_1](\vec{\tau}_2\to\alpha)\}=\Sigma}{\Gamma;\Sigma \vdash h:[\vec{\tau}_1](\vec{\tau}_2\to\alpha)}$$
with α fresh

$$(mabs)\quad \frac{\Gamma, \vec{x}:\vec{\tau};\Sigma \vdash U:\rho}{\Gamma;\Sigma \vdash (\Lambda\vec{x}^{\vec{\tau}}.U):[\vec{\tau}]\rho}$$
with $\Sigma=\emptyset$ and $\rho=\tau'$ or $\Sigma\neq\emptyset$ and $\rho=\vec{\tau}_1\to\alpha$

$$(mapp)\quad \frac{\Gamma;\Sigma \vdash U:[\vec{\tau}]\rho \quad \Gamma \vdash \vec{V}:\vec{\tau}}{\Gamma;\Sigma \vdash U\langle\vec{V}\rangle:\rho}$$
with $\Sigma=\emptyset$ and $\rho=\tau'$ or $\Sigma\neq\emptyset$ and $\rho=\vec{\tau}_1\to\alpha$

$$(habs)\quad \frac{\Gamma;h:[\vec{\tau}_1](\vec{\tau}_2\to\alpha) \vdash U:\vec{\tau}_3\to\alpha}{\Gamma \vdash (\delta h^{[\vec{\tau}_1](\vec{\tau}_2\to\alpha)}.U):\forall\alpha.[\vec{\tau}_1](\vec{\tau}_2\to\alpha)\Rightarrow\vec{\tau}_3\to\alpha}$$

$$(fill)\quad \frac{\Gamma \vdash U:\forall\alpha.[\vec{\tau}_1](\vec{\tau}_2\to\alpha)\Rightarrow\vec{\tau}_3\to\alpha \quad \Gamma \vdash V:([\vec{\tau}_1](\vec{\tau}_2\to\alpha))[\![\alpha:=\tau]\!]}{\Gamma \vdash U\langle V\rangle:(\vec{\tau}_3\to\alpha)[\![\alpha:=\tau]\!]}$$

$$(comp_1)\quad \frac{\Gamma \vdash U:\forall\alpha.[\vec{\sigma}_1](\vec{\sigma}_2\to\alpha)\Rightarrow\vec{\sigma}_3\to\alpha \quad \Gamma \vdash V:[\vec{\sigma}_1](\forall\beta.[\vec{\tau}_1](\vec{\tau}_2\to\beta)\Rightarrow\vec{\sigma}_2\vec{\tau}_3\to\beta)}{\Gamma \vdash U\circ V:\forall\beta.[\vec{\tau}_1](\vec{\tau}_2\to\beta)\Rightarrow\vec{\sigma}_3\vec{\tau}_3\to\beta}$$

$$(comp_2)\quad \frac{\Gamma \vdash U:\forall\alpha.[\vec{\sigma}_1](\vec{\tau}_3\vec{\sigma}_2\to\alpha)\Rightarrow\vec{\sigma}_3\to\alpha \quad \Gamma \vdash V:[\vec{\sigma}_1](\forall\beta.[\vec{\tau}_1](\vec{\tau}_2\to\beta)\Rightarrow\vec{\tau}_3\to\beta)}{\Gamma \vdash U\circ V:\forall\beta.[\vec{\tau}_1](\vec{\tau}_2\vec{\sigma}_2\to\beta)\Rightarrow\vec{\sigma}_3\to\beta}$$

Figure 5.7: Type system for λc^s

easiest way to study types is to study them on normal forms. In Intermezzo 5.1.8 we explained that each (representation of a) segment can be reduced to a segment of the form

$$\lambda\vec{x}.\,[]\,\vec{s}.$$

So, consider the segments C and D given by

$$C \equiv \lambda\vec{x}.\,[]\,\vec{s} \text{ and } \quad D \equiv \lambda\vec{y}.\,[]\,\vec{t}.$$

Let $|\vec{s}| = m$ and $|\vec{y}| = n$. Then the composition of C and D is (see also Figure 5.8)

$$C \circ D = \lambda\vec{x}.\,(\lambda\vec{y}.\,[]\,\vec{t})\,\vec{s}.$$

Note that in $C \circ D$ redexes can be 'matched' between the abstractions over $\vec{y}$ and applications with $\vec{s}$ (here, y_1 matches s_1, y_2 matches s_2 etc.). Hence, an a priori requirement for forming a composition of C and D is that the type of y_i and the type of s_i are the same for $1 \leq i \leq min\{m, n\}$.

In order to determine the normal from of (the representation of) the composition $C \circ D$, the key question is whether $|\vec{y}| = n \geq m = |\vec{s}|$ or $|\vec{s}| = m > n = |\vec{y}|$. If $n \geq m$ then (the representation of) the composition $C \circ D$ reduces to a segment of the form

$$\lambda\vec{x}.\,\lambda y_{n-m}, \ldots, y_n.\,[]\,\vec{t}'.$$

If $m > n$ then (the representation of) the composition $C \circ D$ reduces to a segment of the form

$$\lambda\vec{x}.\,[]\,\vec{t}''s_{m-n} \ldots s_m.$$

Figure 5.9 illustrates these two cases on meta-contexts in lambda calculus. In these contexts the subterms $\vec{s}$ and $\vec{t}$ are annotated with their type.

The two composition rules distinguish the same two cases. We give two examples with concrete types for the compositions rules, one for each. These examples, except for communication, agree with the two examples given for meta-contexts in lambda calculus in Figure 5.9.

example of $(comp_1)$

$$\frac{\Gamma \vdash U \;:\; \forall\beta.[\mathtt{e}]\mathtt{c} \to \beta \Rightarrow \mathtt{f} \to \beta \qquad \Gamma \vdash V \;:\; [\mathtt{e}](\forall\alpha.[\mathtt{a}]\mathtt{b} \to \alpha \Rightarrow \mathtt{c} \to \mathtt{d} \to \alpha)}{\Gamma \vdash U \circ V \;:\; \forall\alpha.[\mathtt{a}]\mathtt{b} \to \alpha \Rightarrow \mathtt{f} \to \mathtt{d} \to \alpha.}$$

example of $(comp_2)$

$$\frac{\Gamma \vdash U \;:\; \forall\beta.[\mathtt{e}]\mathtt{c} \to \mathtt{d} \to \beta \Rightarrow \mathtt{f} \to \beta \qquad \Gamma \vdash V \;:\; [\mathtt{e}](\forall\alpha.[\mathtt{a}]\mathtt{b} \to \alpha \Rightarrow \mathtt{c} \to \alpha)}{\Gamma \vdash U \circ V \;:\; \forall\alpha.[\mathtt{a}]\mathtt{b} \to \mathtt{d} \to \alpha \Rightarrow \mathtt{f} \to \alpha.}$$

In all rules, the basis Γ is used like a basis in the simply typed lambda calculus. This is not the case with Σ, which strictly follows the free hole variable: the basis Σ

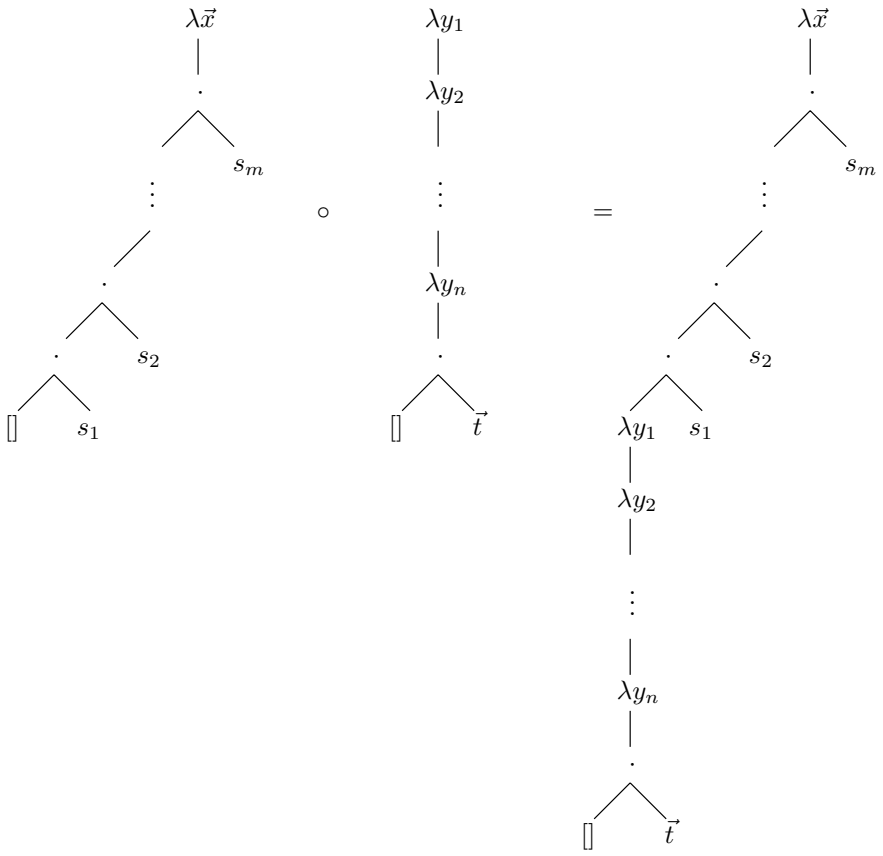


Figure 5.8: Composition of segments

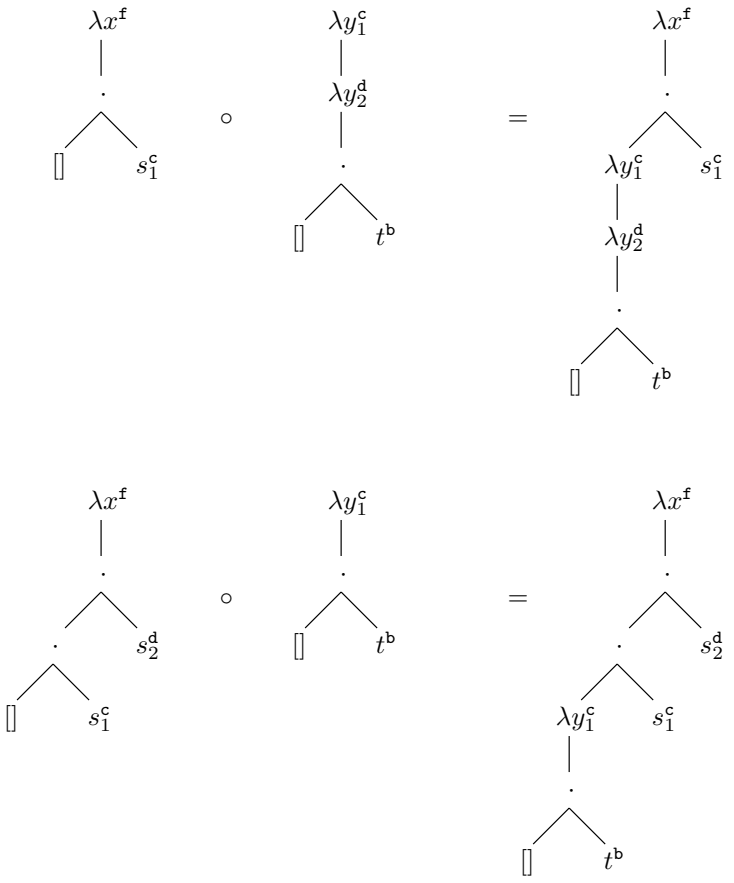


Figure 5.9: Examples of compositions of meta-contexts

changes only by the rules (*hvar*) and (*habs*), where the hole variable is introduced or abstracted; it is empty in rules (*fill*), ($comp_1$) and ($comp_2$), where 'completed' representations of segments are manipulated; and, it is intact in the rest of the rules, where it follows only the left branch of the term when represented as a tree. In this way Σ not only carries information about the type of the free hole variable, but also about the structure of the term. A non-empty Σ means there is a free hole variable on the spine so that the term is an 'incomplete' representation of a segment. This free hole variable is the (only) element of Σ. An empty Σ means that at the end of the spine there is either a variable of a τ-type, or a hole variable bound by a hole abstraction somewhere on the spine. This will be proved later on (see Lemma 5.2.11).

The side conditions of the rules (*abs*), (*app*), (*mabs*) and (*mapp*) are the same, and they allow only two cases: the case where the term U is a representation of a $\lambda^{\rightarrow}$-term, a segment or a function over $\lambda^{\rightarrow}$-terms and segments (i.e. where $\rho = \tau'$ and $\Sigma = \emptyset$), and the case where the term U is a representation of a subterm of a segment (i.e. where $\rho = \vec{\tau} \to \alpha$ and $\Sigma \neq \emptyset$).

The typing rules of λc^s resemble the typing rules of $\lambda c^{\rightarrow}$. One can recognise all but the two last rules ($comp_1$) and ($comp_2$) in the typing rules of $\lambda c^{\rightarrow}$, by ignoring the type variables α and the quantifiers $\forall\alpha$, and by some corrections on Σ. Note that the side conditions are actually extensions of the applicability of a rule, since they involve a ρ-type rather than a τ-type as it is the case in $\lambda c^{\rightarrow}$. The two last rules in λc^s deal with composition. In $\lambda c^{\rightarrow}$ composition is definable, so there are no composition rules. In contrast, in λc^s composition is explicitly present because the terms that mimic composition, such as $\underline{comp}$ of the concluding remark in Section 3.1, are not typable, due to the specific place of holes in segments. Recall that $\underline{comp}$ is the λc-term $\lambda c.\, \delta d.\, \delta g.\, c\lceil \Lambda \vec{u}.\, (d\langle \vec{u} \rangle)\lceil g \rceil\rceil$. Here the variables d and g should have types of the form $[\vec{\tau}_1](\vec{\tau}_2 \to \alpha)$ (i.e. types of hole variables). However, they are not on the spine of a context.

Note that by these typing rules, hole variables are immediately provided with communication, that is, hole variables are labelled. After the rule (*hvar*), which gives a type to a hole variable, the only applicable rule is (*mapp*), because this is the only rule with a term of type $[\vec{\tau}_1]\vec{\tau}_2 \to \alpha$ in the preconditions. This was not the case in the calculi of Chapter 4. Note also that, similarly to the calculus $\lambda c^{\rightarrow}$, only the 'unary' δ, and the binary $\lceil\ \rceil$ and $\circ$ are used.

Example 5.2.4 Some examples of terms that are well-typed are (the types in terms are left out for the sake of readability):

$$b : \mathtt{b}, f : \mathtt{b} \to \mathtt{b} \quad \vdash\ \lambda c^{\forall\alpha.[\mathtt{a}]\alpha \Rightarrow \alpha}.\, c\lceil \Lambda x^{\mathtt{a}}.\, f\rceil\,(c\lceil \Lambda x^{\mathtt{a}}.\, b\rceil) : (\forall\alpha.[\mathtt{a}]\alpha \Rightarrow \alpha) \to \mathtt{b}$$
$$c : \forall\alpha.[\mathtt{a}]\alpha \Rightarrow \mathtt{a} \to \alpha \quad \vdash\ c \circ (\Lambda x^{\mathtt{a}}.\, \delta h^{[\mathtt{a},\mathtt{b}]\beta}.\, \lambda y^{\mathtt{b}}.\, h\,\langle xy \rangle) : \forall\beta.[\mathtt{a},\mathtt{b}]\beta \Rightarrow \mathtt{a} \to \mathtt{b} \to \beta.$$

In the first term, there are two occurrences of the variable c: both are involved in a hole filling but they are filled with terms of different types. This term would not be well-typed without polymorphism.

Some examples of terms that cannot be typed in λc^s are:

- $\delta g.\,(\delta h.\,\lambda x.\,h\langle x\rangle)\lceil g\rceil$: 'renaming' a hole variable h by g is not typable,
- $\delta h.\,(c \circ d)\lceil \Lambda x.\,h\langle x\rangle\rceil$ and $\delta h.\,\lambda y.\,c\lceil \Lambda x.\,h\langle xy\rangle\rceil$: hole filling or composition on the spine of a segment are not typable.

The calculus λc^s is defined on well-typable terms and with rewrite rules that are applicable to these terms.

Definition 5.2.5 (Segment calculus) The terms of λc^s are the well-typed terms of λc according to Definition 5.2.2. The rewrite rules are the rules (β), $(\textcircled{m}\beta)$, (*fill*) and (*comp*) of λc, now restricted to well-typed terms.

i) The lambda calculus rewrite rule is:

$$(\lambda x^{\tau}.\,U)V \to U[\![x := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$\begin{aligned}
&(\Lambda\vec{x}^{\,\vec{\tau}}.\,U)\langle\vec{V}\rangle \to U[\![\vec{x} := \vec{V}]\!] && (\textcircled{m}\beta)\\
&(\delta h^{[\vec{\tau}_1](\vec{\tau}_2\to\alpha)}.\,U)\lceil V\rceil \to U[\![h := V]\!] && (\mathit{fill})\\
&(\delta h^{[\vec{\tau}_1](\vec{\tau}_2\to\alpha)}.\,U)\circ(\Lambda\vec{x}^{\,\vec{\tau}_1}.\,\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\to\beta)}.\,V)\\
&\quad\to \delta g^{[\vec{\sigma}_1](\vec{v}\to\beta)}.\,U[\![h := \Lambda\vec{x}^{\,\vec{\tau}_1}.\,V]\!] && (\circ)
\end{aligned}$$

where the precise form of $\vec{v}$ in the composition rule $(\circ)$ will be specified after the Generation lemma.

Definition 5.2.6 Let the ARS $\mathcal{A}_{\lambda c^s}$ be

$$\mathcal{A}_{\lambda c^s} = \langle \textsc{Ter}(\lambda c^s), \to_\beta, \to_{\textcircled{m}\beta}, \to_{\mathit{fill}}, \to_\circ\rangle.$$

We call the ARS $\mathcal{A}_{\lambda c^s}$ the underlying ARS of the calculus λc^s.

The calculus λc^s is a subsystem of λc

Before we go into the details of the proof that λc^s is a subsystem of λc, we show that λc^s-terms have a unique type. The uniqueness of types follows from the fact that, although some form of polymorphism is present, the polymorphism is built-in and the typing is *à la* Church. By built-in polymorphism we mean that it is encoded in the types by using type variables and $\forall$, as opposed to the polymorphism of $\lambda^{\to}$-Curry, where it is achieved by type substitution (see Lemma 1.2.55(i)).

Proposition 5.2.7 (Uniqueness of types)
If $\Gamma;\Sigma \vdash U : \rho_1$ *and* $\Gamma;\Sigma \vdash U : \rho_2$ *then* $\rho_1 = \rho_2$.

Proof: By induction on the length of the derivation. QED

The calculus λc^s is a subsystem of the framework λc. This is proved in a standard way by showing that the set of terms of λc^s is closed under rewriting. The closure under rewriting is also proved in a standard way via the Generation lemma, Thinning lemma and Substitution lemma. The Thinning lemma holds in λc^s only for Γ, and not for Σ, due to the restriction of Σ to the empty set or singletons and the rigid treatment of Σ in the typing rules. Moreover, such treatment of Σ also makes the proofs of these lemmas a bit more complicated than in a standard case.

Lemma 5.2.8 (Generation lemma)

i) *If* $\Gamma;\Sigma \vdash u : \rho$ *then either* $(u:\rho) \in \Gamma$, *and* $\Sigma = \emptyset$ *or* $\rho = [\vec{\tau}_1](\vec{\tau}_2 \to \alpha)$ *for certain* $\vec{\tau}_1, \vec{\tau}_2 \in \mathcal{T}_s$ *and* $\{(u:\rho)\} = \Sigma$.

ii) *If* $\Gamma;\Sigma \vdash (\lambda x^{\tau}.U) : \rho$ *then either* $\Sigma = \emptyset$ *and there is a* $\tau' \in \mathcal{T}_s$ *such that* $\rho = \tau \to \tau'$ *and* $\Gamma, x:\tau \vdash U : \tau'$, *or* $\Sigma = \{h : [\vec{\sigma}_1](\vec{\sigma}_2 \to \alpha)\}$ *and there are* $\vec{\tau} \in \mathcal{T}_s$ *such that* $\rho = \tau\vec{\tau} \to \alpha$ *and* $\Gamma, x:\tau;\Sigma \vdash U : \vec{\tau} \to \alpha$.

iii) *If* $\Gamma;\Sigma \vdash U_1 U_2 : \rho$ *then either* $\Sigma = \emptyset$, $\rho \in \mathcal{T}_s$, *and there is a* $\tau \in \mathcal{T}_s$ *such that* $\Gamma \vdash U_1 : \tau \to \rho$ *and* $\Gamma \vdash U_2 : \tau$, *or* $\Sigma = \{h : [\vec{\sigma}_1](\vec{\sigma}_2 \to \alpha)\}$ *and* $\rho = \vec{\tau} \to \alpha$ *for* $\vec{\tau} \in \mathcal{T}_s$ *and* $\Gamma;\Sigma \vdash U_1 : \tau\vec{\tau} \to \alpha$ *and* $\Gamma \vdash U_2 : \tau$.

iv) *If* $\Gamma;\Sigma \vdash (\Lambda\vec{x}^{\vec{\tau}}.U) : \rho$ *then either* $\Sigma = \emptyset$ *and there is a* $\tau \in \mathcal{T}_s$ *such that* $\rho = [\vec{\tau}]\tau$ *and* $\Gamma, \vec{x} : \vec{\tau} \vdash U : \tau$, *or* $\Sigma = \{h : [\vec{\sigma}_1](\vec{\sigma}_2 \to \alpha)\}$ *and there is* $\vec{\tau}_1 \in \mathcal{T}_s$ *such that* $\rho = [\vec{\tau}](\vec{\tau}_1 \to \alpha)$ *and and* $\Gamma, \vec{x} : \vec{\tau};\Sigma \vdash U : \vec{\tau}_1 \to \alpha$.

v) *If* $\Gamma;\Sigma \vdash U\langle\vec{U}\rangle : \rho$ *then either* $\Sigma = \emptyset$, $\rho \in \mathcal{T}_s$ *and there are* $\vec{\tau} \in \mathcal{T}_s$ *such that* $\Gamma \vdash U : [\vec{\tau}]\rho$ *and* $\Gamma \vdash \vec{U} : \vec{\tau}$, *or* $\Sigma = \{h : [\vec{\sigma}_1](\vec{\sigma}_2 \to \alpha)\}$, *there are* $\vec{\tau}_1, \vec{\tau}_2 \in \mathcal{T}_s$ *such that* $\rho = \vec{\tau}_2 \to \alpha$, $\Gamma;\Sigma \vdash U : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha)$ *and* $\Gamma \vdash \vec{U} : \vec{\tau}_1$.

vi) *If* $\Gamma;\Sigma \vdash (\delta h^{[\vec{\tau}_1](\vec{\tau}_2 \to \alpha)}.U) : \rho$ *then* $\Sigma = \emptyset$ *and there are* $\vec{\tau}_3 \in \mathcal{T}_s$ *such that* $\rho = \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha$ *and* $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U : \vec{\tau}_3 \to \alpha$.

vii) *If* $\Gamma;\Sigma \vdash U_1\lceil U_2\rceil : \rho$ *then* $\Sigma = \emptyset$ *and there are* $\tau, \vec{\tau}_1, \vec{\tau}_2, \vec{\tau}_3 \in \mathcal{T}_s$ *and* $\alpha \in TV$ *such that* $\rho = \vec{\tau}_3 \to \tau$, $\Gamma \vdash U_1 : \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha$ *and* $\Gamma \vdash U_2 : [\vec{\tau}_1](\vec{\tau}_2 \to \tau)$.

viii) *If* $\Gamma;\Sigma \vdash U \circ V : \rho$ *then* $\Sigma = \emptyset$ *and there are* $\vec{\tau}_1, \vec{\tau}_2, \vec{\tau}_3, \vec{\sigma}_1, \vec{\sigma}_2, \vec{\sigma}_3 \in \mathcal{T}_s$ *and* $\alpha, \beta \in TV$ *such that either*

$$\begin{aligned} &\rho = \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_3\vec{\sigma}_3 \to \beta \\ &\Gamma \vdash U : \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha \\ &\Gamma \vdash V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_2\vec{\sigma}_3 \to \beta), \end{aligned}$$

or

$$\begin{aligned} &\rho = \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \Rightarrow \vec{\tau}_3 \to \beta \\ &\Gamma \vdash U : \forall\alpha.[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha \\ &\Gamma \vdash V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\sigma}_3 \to \beta). \end{aligned}$$

Proof: Suppose $\Gamma;\Sigma \vdash U : \rho$. The statements follow by distinguishing the cases of the structure of U. QED

Using the Generation lemma, two precise forms of the rule $\circ$ can be specified: if $V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_2\vec{\sigma}_3 \to \beta)$ then

$$(\delta h^{[\vec{\tau}_1](\vec{\tau}_2 \to \alpha)}.U) \circ (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta)}.V) \;\to_{\circ}\; \delta g^{[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta)}.U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V]\!],$$

and if $V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\sigma}_3 \to \beta)$ then

$$(\delta h^{[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2\to\alpha)}.U) \circ (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1]\vec{\sigma}_2\to\beta}.V) \;\to_\circ\; \delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2\to\beta)}.U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V]\!]\,.$$

Lemma 5.2.9 (Thinning for Γ) *Let $\Gamma;\Sigma \vdash U : \rho$. Then $\Gamma';\Sigma \vdash U : \rho$ for all Γ' with $\Gamma \subseteq \Gamma'$.*

Proof: The statement is proved by the induction to the length of the derivation. QED

As already mentioned above, Thinning lemma does not hold for the basis Σ. In fact, the basis Σ may not be enlarged nor can it be reduced to form the same derivation. There is a strong connection between the form of a well-typed λc-term U, its type ρ and the base Σ.

Lemma 5.2.10 *If $\Gamma;\Sigma \vdash U : \rho$ then $\Sigma = \emptyset$ or $\Sigma = \{h : [\vec{\tau}_1]\vec{\tau}_2 \to \alpha\}$.*

Proof: The statement is proved by induction to the length of the derivation. QED

Lemma 5.2.11 *Let $\Gamma;\Sigma \vdash U : \rho$. Then the following statements are equivalent:*

i) $\Sigma = \{h : [\vec{\tau}_1]\vec{\tau}_2 \to \alpha\}$;

ii) hole variable h occurs free on the spine of the term U;

iii) $\rho = [\vec{\sigma}_1]\vec{\sigma}_2 \to \alpha$ *or* $\rho = \vec{\sigma}_2 \to \alpha$.

In particular, this lemma implies that a composition or a hole filling cannot occur on the spine of a segment. For example, suppose $U \equiv \delta h^\rho.U'$ with a composition in the spine. Then, according to the typing rule (*habs*), U' should be typable with $\Sigma = \{(h : \rho)\}$. That means also that U' and all its subterms with the root on the spine of U', including the composition, must be typable with $\Sigma = \{(h : \rho)\}$. But, according to the composition typing rules, compositions are typable only with an empty Σ.

The Substitution lemma deals with type and term substitutions that arise in a rewrite step. In the calculi of Chapter 4, the corresponding lemmas deal with type-preserving substitutions for terms, that is, with the substitutions $[\![\vec{x} := \vec{V}]\!]$ where x_i and V_i are a variable and a term of the same type, for $1 \le i \le |\vec{x}| = |\vec{V}|$. However, in the rewrite steps of λc^s also substitutions for type variables may arise, and some terms substitutions may arise which are not type-preserving in a classical sense. These substitutions can be classified as follows:

- the substitutions $[\![\alpha := \rho]\!]$ where $\rho = \tau$ or $\rho = \vec{\sigma} \to \beta$, and
- the substitutions $[\![h := V]\!]$ where $h : [\vec{\tau}_1]\vec{\tau}_2 \to \alpha$ and $V : [\vec{\tau}_1]\vec{\tau}_2 \to \rho$ where $\rho = \tau$ or $\rho = \vec{\sigma} \to \beta$.

Next to these substitutions there are of course the type-preserving term substitutions.

For the proof the Substitution lemma, we need a small lemma about replacing a hole variable h by another hole variable g with a type that fits the type of the original hole variable h.

Lemma 5.2.12
If $\Gamma; h : [\vec{\sigma}_1](\vec{\sigma}_2 \to \alpha) \vdash U : \rho$ *then* $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\sigma} \to \beta) \vdash U[\![h := g]\!] : \rho[\![\alpha := \vec{\sigma} \to \beta]\!]$.

Proof: By abbreviating the type of h by ρ_α, this statement can be given a more readable form:

> Let $\Gamma; h : \rho_\alpha \vdash U : \rho$. Then $\Gamma; g : \rho_\alpha\,[\![\alpha := \vec{\sigma} \to \beta]\!] \vdash U[\![h := g]\!] : \rho[\![\alpha := \vec{\sigma} \to \beta]\!]$.

The proof proceeds by induction to the structure of U and uses Corollary 5.2.11.
QED

Lemma 5.2.13 (Substitution lemma)

i) *If* $\Gamma, \vec{x} : \vec{\tau}; \Sigma \vdash U : \rho$ *and* $\Gamma \vdash \vec{V} : \vec{\tau}$ *then* $\Gamma; \Sigma \vdash U[\![\vec{x} := \vec{V}]\!] : \rho$.

ii) *If* $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U : \rho$ *and* $\Gamma \vdash V : [\vec{\tau}_1](\vec{\tau}_2 \to \tau)$ *then* $\Gamma \vdash U[\![h := V]\!] : \rho[\![\alpha := \tau]\!]$.

iii) *If* $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U : \rho$ *and* $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : [\vec{\tau}_1](\vec{\tau}_2\vec{\sigma}_3 \to \beta)$ *then* $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash U[\![h := V]\!] : (\rho[\![\alpha := \vec{\sigma}_3 \to \beta]\!])$.

iv) *If* $\Gamma; h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \vdash U : \rho$ *and* $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : [\vec{\tau}_1](\vec{\sigma}_3 \to \beta)$ *then* $\Gamma; k : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \vdash U[\![h := V[\![g := k]\!]]\!] : \rho[\![\alpha := \beta]\!]$.

Proof: The substitutions in (i) and the substitution $[\![h := V[\![g := k]\!]]\!]$ in (iv) are type-preserving term substitutions, the other substitutions fall under the classification as given above. All proofs are conducted by induction to U, and by using the Generation lemma and the Thinning lemma for Γ. We show only the difficult cases, namely the parts (iii) and (iv).

Proof of (iii):

($U \equiv u$)**:** Because the hole variables basis is not empty, the variable u must be the hole variable, that is, $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha)$. Then $h[\![h := V]\!] = V$, and $\rho[\![\alpha := \vec{\sigma}_3 \to \beta]\!] = [\vec{\tau}_1](\vec{\tau}_2\vec{\sigma}_3 \to \beta)$. We have $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : [\vec{\tau}_1](\vec{\tau}_2\vec{\sigma}_3 \to \beta)$ by the assumption, as requested.

($U \equiv U_1U_2$)**:** Then, $\rho = \vec{v} \to \alpha$ for $\vec{v} \in \mathcal{T}_s$ and there is $v \in \mathcal{T}_s$ such that and $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U_1 : v\vec{v} \to \alpha$ and $\Gamma \vdash U_2 : v$. By the induction hypothesis, $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash U_1[\![h := V]\!] : v\vec{v}\vec{\sigma}_3 \to \beta$. Note that the variable h does not occur free in U_2, so $U_2[\![h := V]\!] = U_2$ and $\Gamma \vdash U_2[\![h := V]\!] : v$. Then by the rule (app) and the definition of substitution, $\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash (U_1U_2)[\![h := V]\!] : \vec{v}\vec{\sigma}_3 \to \beta$.

The cases where $U \equiv \lambda x^{\tau}.U'$ and $U \equiv \Lambda\vec{x}^{\vec{\tau}}.U'$ are straightforward applications of the induction hypothesis. The case where $U \equiv U'\langle\vec{U}\rangle$ is similar to the application case above. The cases where $U \equiv \delta k^{\rho'}.U'$, $U \equiv U_1\lceil U_2\rceil$ and $U \equiv U_1 \circ U_2$ do not occur, because these cases require an empty Σ (Lemma 5.2.11).

Proof of (*iv*): We show only two difficult cases; other cases are treated in the same way as in (*iii*).

($U \equiv u$)**:** Because the hole variables basis is not empty, the variable u must be the hole variable, that is, $\Gamma; h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \vdash h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha)$. Then $h[\![h := V[\![g := k]\!]]\!] = V[\![g := k]\!]$.

By Lemma 5.2.12 we have $\Gamma; k : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \vdash V[\![g := k]\!] : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \beta)$, as requested.

($U \equiv U_1U_2$)**:** Then, $\rho = \vec{v} \to \alpha$ for $\vec{v} \in \mathcal{T}_s$ and there is $\upsilon \in \mathcal{T}_s$ such that and $\Gamma; h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \vdash U_1 : \upsilon\vec{v} \to \alpha$ and $\Gamma \vdash U_2 : \upsilon$. By the induction hypothesis, $\Gamma; k : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \vdash U_1[\![h := V[\![g := k]\!]]\!] : \upsilon\vec{v} \to \beta$. Note that the variable h does not occur free in U_2, so $U_2[\![h := V[\![g := k]\!]]\!] = U_2$ and $\Gamma \vdash U_2[\![h := V[\![g := k]\!]]\!] : \upsilon$. Then by the rule (*app*) and the definition of substitution, $\Gamma; k : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \vdash (U_1U_2)[\![h := V[\![g := k]\!]]\!] : \vec{v} \to \beta$.

QED

Proposition 5.2.14 (Subject reduction)
If $\Gamma; \Sigma \vdash U : \rho$ *and* $U \twoheadrightarrow V$*, then* $\Gamma; \Sigma \vdash V : \rho$.

Proof: The proof is conducted as follows. One first shows that contractions $L \to R$ of all rules preserve types. Then, one shows that any one-step reduction $C[L] \to C[R]$ preserves types by induction to C. Finally, the statement is proved by induction to the length of the reduction.

We show only the most difficult part of the proof, namely that the contraction of a composition redex preserves types. So, let $L \to R$ be a contraction of the rule ($\circ$). According to the Generation lemma we have two cases.

Case 1: We have the following derivations

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U : \vec{\tau}_3 \to \alpha}{\Gamma \vdash (\delta h^{[\vec{\tau}_1](\vec{\tau}_2 \to \alpha)}.U) : \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha} \quad \vdots \quad \Gamma, \vec{x} : \vec{\tau}_1; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : \vec{\tau}_2\vec{\sigma}_3 \to \beta}{\vdots \quad \Gamma, \vec{x} : \vec{\tau}_1 \vdash (\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta)}.V) : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_2\vec{\sigma}_3 \to \beta}}{\vdots \quad \Gamma \vdash (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta)}.V) : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_2\vec{\sigma}_3 \to \beta)}}{\begin{array}{c}\Gamma \vdash (\delta h^{[\vec{\tau}_1](\vec{\tau}_2 \to \alpha)}.U) \circ (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta)}.V) \\ : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_3\vec{\sigma}_3 \to \beta.\end{array}}$$

Then, by the rule (*mabs*)

$$\frac{\Gamma, \vec{x} : \vec{\tau}_1; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : \vec{\tau}_2\vec{\sigma}_3 \to \beta}{\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash (\Lambda\vec{x}^{\vec{\tau}_1}.V) : [\vec{\tau}_1](\vec{\tau}_2\vec{\sigma}_3 \to \beta).}$$

Then, because $\Gamma; h : [\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \vdash U : \vec{\tau}_3 \to \alpha$ and by the Substitution lemma (*iii*),

$$\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V]\!] : \vec{\tau}_3\vec{\sigma}_3 \to \beta.$$

Finally, by the rule (*habs*),

$$\begin{array}{l}\Gamma \vdash (\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\to\beta)}.U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V]\!]) \\ \qquad : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_3\vec{\sigma}_3 \to \beta\end{array}$$

as requested.

Case 2: We have the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \vdash U : \vec{\tau}_3 \to \alpha}{\Gamma \vdash (\delta h^{[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2\to\alpha)}.U) : \forall\alpha.[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha} \\ \vdots \quad \Gamma, \vec{x} : \vec{\tau}_1; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : \vec{\sigma}_3 \to \beta}{\vdots \quad \Gamma, \vec{x} : \vec{\tau}_1 \vdash (\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\to\beta)}.V) : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\sigma}_3 \to \beta}}{\vdots \quad \Gamma \vdash (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\to\beta)}.V) : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\sigma}_3 \to \beta)}}{\begin{array}{l}\Gamma \vdash (\delta h^{[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2\to\alpha)}.U) \circ (\Lambda\vec{x}^{\vec{\tau}_1}.\delta g^{[\vec{\sigma}_1](\vec{\sigma}_2\to\beta)}.V) \\ \qquad : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \Rightarrow \vec{\tau}_3 \to \beta.\end{array}}$$

Then, by the rule (*mabs*)

$$\frac{\Gamma, \vec{x} : \vec{\tau}_1; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash V : \vec{\sigma}_3 \to \beta}{\Gamma; g : [\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \vdash (\Lambda\vec{x}^{\vec{\tau}_1}.V) : [\vec{\tau}_1](\vec{\sigma}_3 \to \beta).}$$

Then, because $\Gamma; h : [\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \vdash U : \vec{\tau}_3 \to \alpha$ and by the Substitution lemma (*iii*),

$$\Gamma; k : [\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \vdash U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V[\![g := k]\!]]\!] : \vec{\tau}_3 \to \beta.$$

Finally, by the rule (*habs*),

$$\begin{array}{l}\Gamma \vdash (\delta k^{[\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2\to\beta)}.U[\![h := \Lambda\vec{x}^{\vec{\tau}_1}.V[\![g := k]\!]]\!]) \\ \qquad : \forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2\vec{\tau}_2 \to \beta) \Rightarrow \vec{\tau}_3 \to \beta\end{array}$$

as requested. QED

The closure of the set of well-typed terms implies that the calculus λc^s satisfies the definition of a subsystem (Definition 1.1.14) of the framework λc.

Theorem 5.2.15 (Subsystem λc^s) *The underlying ARS $\mathcal{A}_{\lambda c^s}$ of the calculus λc^s is an indexed sub-ARS of the underlying ARS $\mathcal{A}_{\lambda c}$ of the context calculus λc.*

Proof: The statement is proved using the Subject reduction lemma 5.2.14. QED

Properties of rewriting in λc^s

Rewriting in λc^s is complete, that is, confluent and strongly normalising.

Theorem 5.2.16 (Confluence) *The calculus λc^s is confluent.*

Proof: This property follows by using Lemma 1.1.10, the commutation property of each pair of rewrite rules of the framework λc (Theorem 3.2.38) and the fact that λc^s is a subsystem of λc (Theorem 5.2.15). QED

Rewriting in λc^s is strongly normalising. The proof proceeds via translation to $\lambda 2$ (see Section 1.2.3). The strong normalisation of λc^s follows from the strong normalisation of $\lambda 2$.

Definition 5.2.17 (Translation of λc^s to $\lambda 2$)

i) Define $\llbracket\ \rrbracket : \mathcal{P}_s \to \textsc{Typ}(\lambda 2)$ as a function that translates the types to the types of $\lambda 2$:

$$\begin{array}{lcl}
\llbracket\ \mathtt{a}\ \rrbracket & = & \mathtt{a} \\
\llbracket \tau \to \tau' \rrbracket & = & \llbracket \tau \rrbracket \to \llbracket \tau' \rrbracket \\
\llbracket \forall\alpha.[\vec{\tau}_1]\vec{\tau}_2 \to \alpha \Rightarrow \vec{\tau}_3 \to \alpha \rrbracket & = & \forall\alpha.(\llbracket \vec{\tau}_1 \rrbracket \llbracket \vec{\tau}_2 \rrbracket \to \alpha) \to (\llbracket \vec{\tau}_3 \rrbracket \to \alpha) \\
\llbracket \vec{\tau} \to \alpha \rrbracket & = & \llbracket \vec{\tau} \rrbracket \to \alpha \\
\llbracket [\vec{\tau}]\tau \rrbracket & = & \llbracket \vec{\tau} \rrbracket \to \llbracket \tau \rrbracket \\
\llbracket [\vec{\tau}_1]\vec{\tau}_2 \to \alpha \rrbracket & = & \llbracket \vec{\tau}_1 \rrbracket \llbracket \vec{\tau}_2 \rrbracket \to \alpha.
\end{array}$$

ii) Define $\llbracket\ \rrbracket : \textsc{Ter}(\lambda c^s) \to \textsc{Ter}(\lambda 2)$ as

$$\begin{array}{lcl}
\llbracket\ u \rrbracket & = & u \\
\llbracket\ \lambda x^{\tau}.U\ \rrbracket & = & \lambda x^{\llbracket \tau \rrbracket}.\llbracket U \rrbracket \\
\llbracket\ U V\ \rrbracket & = & \llbracket U \rrbracket \llbracket V \rrbracket \\
\llbracket\ \Lambda\vec{x}^{\vec{\tau}}.U\ \rrbracket & = & \lambda\vec{x}^{\llbracket \vec{\tau} \rrbracket}.\llbracket U \rrbracket \\
\llbracket\ U\langle\vec{U}\rangle\ \rrbracket & = & \llbracket U \rrbracket \llbracket U_1 \rrbracket \ldots \llbracket U_n \rrbracket \\
\llbracket\ \delta h^{[\vec{\tau}_1]\vec{\tau}_2 \to \alpha}.U\ \rrbracket & = & \Lambda\alpha.\,\lambda h^{\llbracket [\vec{\tau}_1]\vec{\tau}_2 \to \alpha \rrbracket}.\llbracket U \rrbracket \\
\llbracket\ U\lceil V \rceil\ \rrbracket & = & \llbracket U \rrbracket \llbracket \tau \rrbracket \llbracket V \rrbracket
\end{array}$$

if, for certain Γ,

$$\Gamma \vdash U : \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha$$
$$\Gamma \vdash V : ([\vec{\tau}_1](\vec{\tau}_2 \to \alpha))[\![\alpha := \tau]\!]$$

$$\llbracket\ U \circ V\ \rrbracket \quad = \quad comp_1 \llbracket U \rrbracket \llbracket V \rrbracket$$

if, for certain Γ (the types of c and d are the types of U and V, respectively),
$\Gamma \vdash U : \forall\alpha.[\vec{\tau}_1](\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha$
$\Gamma \vdash V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\tau}_2\vec{\sigma}_3 \to \beta)$
$comp_1 \equiv \lambda c.\, \lambda d.\, \Lambda\beta.\lambda g^{\llbracket \vec{\sigma}_1 \rrbracket \llbracket \vec{\sigma}_2 \rrbracket \to \beta}.\, c(\llbracket \vec{\sigma}_3 \rrbracket \to \beta)(\lambda \vec{x}^{\llbracket \vec{\tau}_1 \rrbracket}.\, d\vec{x}\beta g)$

$\llbracket U \circ V \rrbracket \quad = \quad comp_2 \llbracket U \rrbracket \llbracket V \rrbracket$

if, for certain Γ (the types of c and d are the types of U and V, respectively),
$\Gamma \vdash U : \forall\alpha.[\vec{\tau}_1](\vec{\sigma}_3\vec{\tau}_2 \to \alpha) \Rightarrow \vec{\tau}_3 \to \alpha$
$\Gamma \vdash V : [\vec{\tau}_1](\forall\beta.[\vec{\sigma}_1](\vec{\sigma}_2 \to \beta) \Rightarrow \vec{\sigma}_3 \to \beta)$
$comp_2 \equiv \lambda c.\, \lambda d.\, \Lambda\beta.\lambda g^{\llbracket \vec{\sigma}_1 \rrbracket \llbracket \vec{\sigma}_2 \rrbracket \llbracket \vec{\tau}_2 \rrbracket \to \beta}.\, c\beta(\lambda \vec{x}^{\llbracket \vec{\tau}_1 \rrbracket}.\, d\vec{x}(\llbracket \vec{\tau}_2 \rrbracket \to \beta) g)$.

iii) Let Δ be a basis. Then $\llbracket \Delta \rrbracket = \{(u : \llbracket \rho \rrbracket) \mid (u : \rho) \in \Delta\}$.

Translation preserves typing.

Proposition 5.2.18 *If* $\Gamma; \Sigma \vdash_{\lambda c^s} U : \rho$ *then* $\llbracket \Gamma \rrbracket \cup \llbracket \Sigma \rrbracket \vdash_{\lambda 2} \llbracket U \rrbracket : \llbracket \rho \rrbracket$.

Proof: The proof is done by induction to the length of $\Gamma; \Sigma \vdash_{\lambda c^s} U : \rho$. One checks the typing rules of λc^s: from the translations of the premisses, the translations of the conclusions can be derived in $\lambda 2$. Then each derivation step in $\Gamma; \Sigma \vdash_{\lambda c^s} U : \rho$ can be translated into one or more derivation steps in $\lambda 2$. QED

Translation preserves rewrite steps.

Proposition 5.2.19 *If* $\Gamma; \Sigma \vdash_{\lambda c^s} U : \rho$ *and* $U \to V$ *in* λc^s, *then* $\llbracket U \rrbracket \twoheadrightarrow \llbracket V \rrbracket$ *in* $\lambda 2$.

Proof: In general, the rewrite steps in λc^s are translated into many β-steps (involving both λ-abstractions and Λ-abstractions), with one exception: a $\textcircled{m}\beta$-step where the multiple abstraction and the multiple application are empty, that is, $(\Lambda\epsilon.\, U)\langle\rangle \to_{\textcircled{m}\beta} U$, which in translation results in an empty β-step: $\llbracket (\Lambda\epsilon.\, U)\langle\rangle \rrbracket = U = \llbracket U \rrbracket$. QED

Theorem 5.2.20 (Strong normalisation) *Rewriting in* λc^s *is strongly normalising.*

Proof: Let $U_0 \in \textsc{Ter}(\lambda c^s)$ and suppose r is an infinite rewrite sequence in λc^s:

$r : \quad U_0 \to U_1 \to U_2 \to \ldots \ (\infty)$.

Note indeed that if U_0 is a λc^s-term, then so are all its reducts. Then, the translation of $U_0, U_1, U_2, \ldots$ to $\lambda 2$ results in a rewrite sequence $\llbracket r \rrbracket$ in the simply typed lambda calculus:

$\llbracket r \rrbracket : \quad \llbracket U_0 \rrbracket \twoheadrightarrow \llbracket U_1 \rrbracket \twoheadrightarrow \llbracket U_2 \rrbracket \twoheadrightarrow \ldots \ (\infty)$.

Because there are no infinite rewrite sequences in $\lambda 2$, the tail of $\llbracket r \rrbracket$ must eventually be empty, i.e. $\llbracket U_n \rrbracket \equiv \llbracket U_{n+1} \rrbracket \equiv \ldots$. These steps can only be translations of 'empty' $\textcircled{m}\beta$-steps, i.e. $U_n \equiv C[(\Lambda\epsilon.\, U)\langle\rangle] \to_{\textcircled{m}\beta} C[U] \equiv U_{n+1} \ldots$. However, since λc^s-terms are finite, there cannot be infinitely many such steps starting from U_n. QED

Corollary 5.2.21 *Rewriting in* λc^s *is complete.*

Comparison to de Bruijn's calculus and $\lambda c^{\rightarrow}$

We compare the calculus λc^s to de Bruijn's segment calculus. Both calculi are extensions of lambda calculus with segments, but there are a couple of significant differences.

The first kind of differences lies in the choice for the lambda calculus notation: using name-carrying variables as in λc^s or using name-free variables as in de Bruijn's calculus. Related to the notation is the way of implementing β-rewrite relation. In our calculus the β-rewriting is defined using meta-substitution, whereas in de Bruijn's calculus β-rewriting is defined stepwise by traversing the pair $(\lambda x.\_)t$ through a term or a segment s in the redex $(\lambda x.\, s)t$.

In addition, there are differences in the way segments are represented and used. These differences pertain to the representation of a hole, representation of a segment and representation of segment variables; they are discussed in turn.

- In our calculus holes are denoted by a hole variable h whereas in de Bruijn's calculus holes are denoted by a constant ω. Moreover, in λc^s hole variables are 'labelled' by terms, viz. $h\langle\vec{V}\rangle$; the terms in the 'label' capture the effects of β-step within a segment. In de Bruijn's calculus holes are labelled by reference transforming mappings, which can only adjust the intended bindings, and cannot capture the effects of a β-step within a segment. For this reason, the redex in a segment keeps 'hanging' in front of the hole.

- In our calculus a segment is an abstraction over a hole variable $\delta h.U$, where δh explicitly denotes the root of the segment, while in de Bruijn's calculus the root of a segment is implicilty determined.

- In our calculus segment variables are ordinary nullary variables (e.g. c in $c \circ (\Lambda x.\, \delta h.\, \lambda y.\, h\langle x, y\rangle)$), whereas in de Bruijn's calculus segment variables are unary (e.g. $\eta(1,5)$ in $\lambda\eta(1,5)\omega(id_2)$). In de Bruijn's calculus, whenever a segment variable is used, it is immediately provided with an object to be filled into the hole of the segment for which this variable holds place. With such unary segment variables, the spine of a term is defined differently in de Bruijn's calculus and our calculus. For example, the end of the spine in the term $\eta(1,5)\lambda\omega(id_2)$ is $\omega(id_2)$, so this term is a segment. In the representation of the same term in λc^s, in the term $c \circ (\Lambda x.\, \delta h.\, \lambda y.\, h\langle x, y\rangle)$, the end of the spine is c, so this term is not a segment (although this term does result in a segment after a substitution for c and reduction to normal form). This implies that, with such segment variables, de Bruijn's calculus includes a more flexible notion of segment than our calculus. In particular, it allows composition or hole filling on the spine of a segment, which is not allowed in our calculus.

- In de Bruijn's and Balsters' typing, the type of the hole is not explicitly fixed, but it is computed by the typing function. The type of a hole is computed in such a way that if $\lambda\vec{x}^{\vec{\tau}}.\, []\,\vec{s}$ is a segment, then the type of $[]\,\vec{s}$ is a base type. In our calculus the type of $[]\,\vec{s}$ may also be functional.

If we restrict the base type set to a singleton, say $\{\mathtt{a}\}$, the types in λc^s are comparable to the frames in De Bruijn's version. In λc^s the type of a segment is also described by three sequences of types (i.e. types of the unmatched abstractions in front, types of the unmatched applications at the hole and types of the communication). For example, the segment type $\forall\alpha.[\mathtt{a}]\mathtt{a} \to \mathtt{a} \to \alpha \Rightarrow \mathtt{a} \to \mathtt{a} \to \mathtt{a} \to \alpha$ corresponds to the frame $(\mathtt{a}, \mathtt{a}, \mathtt{a};\ \mathtt{a}, \mathtt{a};\ \mathtt{a})$ in de Bruijn's calculus.

In Section 4.2 an another context calculus was given for simply typed lambda calculus, namely the calculus $\lambda c^{\to}$. The calculus $\lambda c^{\to}$ included contexts without restrictions on the number and position of hole, and it allowed no polymorphism for context types. However, technically, the calculus λc^s resembles $\lambda c^{\to}$. We have already made many comparisons throughout this section in types and type system. The main difference is in the presence and usage of type variables. By dropping the polymorphism in λc^s and adjusting the treatment of Σ, one obtains the calculus $\lambda c^{\to}$. Another difference is in the specific position of the hole in λc^s and the treatment of the base Σ in the type system. Moreover, composition is explicitly present in λc^s whereas in $\lambda c^{\to}$ it is definable.

Some (non-)examples in λc^s

The introductory example cannot be represented within λc^s, because it involves λ-terms that are not typable in $\lambda^{\to}$, thus also not typable in λc^s. In the example, it holds $C[M] \to_\beta x\,x$, and the term $x\,x$ is not typable in λc^s.

The (representation of) context $C \equiv (\lambda y.\,[])\,x$ itself is typable in λc^s (note that C is a segment):

$$x : \mathtt{a} \vdash \delta h^{[\mathtt{a}]\alpha}.(\lambda y^{\mathtt{a}}.\,h\,\langle y\rangle)\,x : \forall\alpha.[\mathtt{a}]\alpha \Rightarrow \alpha.$$

Note that for example, the big example of Figure 5.4 is not typable in λc^s, because this example requires dependent types. This example will be typable in the next chapter.

Summary of comparisons

We add the calculus λc^s into the figures of Section 4.4. The meaning of arrows is the same in that section.

Figure 5.10 summarises the relationship between the framework λc on the one hand, and the calculi λc^{λ}, $\lambda c^{\to}$, $\lambda c^{\cong}$ and λc^s on the other hand.

Table 5.1 summarises the expressivity of the applications of the context calculus λc that are considered in this thesis.

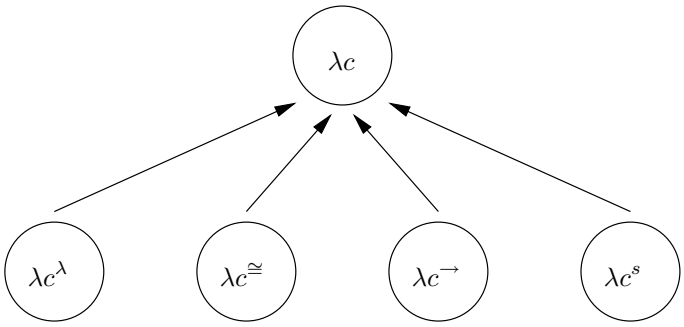


Figure 5.10: Relationship with λc and its applications

calculus	formalises	functions over contexts?	translated to
λc^{λ}	contexts of the untyped λ-calculus	no	untyped λ-calculus
$\lambda c^{\rightarrow}$	contexts of $\lambda^{\rightarrow}$	yes	$\lambda^{\rightarrow}$
$\lambda c^{\cong}$	contexts of the untyped λ-calculus	yes	untyped λ-calculus
λc^{s}	segments of $\lambda^{\rightarrow}$	yes	$\lambda 2$

Table 5.1: The expressivity of the context calculi

Chapter 6

The context cube $\lambda[\,]$: the lambda cube with contexts

In the preceding chapters we have considered lambda calculi with contexts, where a particular lambda calculus was either untyped or simply typed, and where contexts were defined as *terms* with holes. In this chapter we aim to gain more expressive power by considering also lambda calculi with dependent types and by allowing holes to occur also in types.

We present here the context cube, or the $\lambda[\,]$-cube for short, which is a collection of eight context calculi related to the eight systems of Barendregt's lambda cube with contexts. Indirectly, because contexts for the lambda cube are difficult to describe by using informal notation as in the lambda calculus, the $\lambda[\,]$-cube defines a notion of context for the lambda cube. The novelty of the $\lambda[\,]$-cube is that it includes also context calculi dealing with dependent types. The simplicity of the $\lambda[\,]$-cube lies in the fact that it can be translated into the lambda cube.

This chapter is structured as follows. Section 6.1 recalls Barendregt's lambda cube and discusses the problems when defining and formalising its meta-contexts. Section 6.2 is a gentle introduction to the context cube. It describes what the context cube is and the approach to formalisation of meta-contexts in the context cube. Section 6.3 contains the definition of the expressions, rewrite relations and typing in the systems of the context cube. Section 6.4 shows that rewriting in the $\lambda[\,]$-cube is confluent and strongly normalising, and that each system $\lambda[S]$ of the $\lambda[\,]$-cube can be translated into the corresponding system λS of the lambda cube. Section 6.5 illustrates how the context cube can be employed for representing de Bruijn's segments with 'polymorphic types' and how such segments can be used for representing mathematical structures in proof checking. The fact that the context cube can be translated into the lambda cube entails that our representations of mathematical structures could be coded into the existing tools for automated reasoning that are

based on the lambda cube or type theories.

In general, in this chapter we concentrate on the expressivity related to contexts, and do not consider the expressivity of a particular system $\lambda[S]$ of the $\lambda[\,]$-cube, because the latter is basically the same as in the system λS of the lambda cube.

6.1 Barendregt's lambda cube

In this section we give the background for our $\lambda[\,]$-cube by analysing the lambda cube.

The lambda cube, or the λ-cube for short, which was introduced by H.P. Barendregt, comprises eight systems of typed lambda calculi, all of which are typed à la Church. Most of the eight systems were already known systems at the time, whether in the same format as in the λ-cube or in an equivalent format. However, the ingenuity of the λ-cube is that it presents the eight systems in a way that allows for uniform development of meta-theory. This description method has been further generalised to Pure Type Systems (PTSs) (see for example [Ber88, Ter89, GN91]).

The systems of the λ-cube are parametrised by the sorts of dependencies that may occur between terms and types.

We explain what is here meant by the dependency between terms and types. In the calculi and the typing systems of the preceding chapters, terms and types were defined as two different syntactic categories. Take for example the simply typed lambda calculus[1] $\lambda^{\to}$ as defined in Section 1.2.2. In $\lambda^{\to}$, first types are defined and then terms, where abstractions are annotated with types. Furthermore, terms and types are only related by the typing rules in typing statements $\Gamma \vdash M : \tau$, where M stands for a term and τ for a type. In contrast, in the λ-cube, terms and types may depend on each other. An example of a term depending on types is the term $\Lambda\alpha.\, \lambda x : \alpha.\, x$, which explicitly depends on the type variable α. As a consequence of such dependencies, in the λ-cube terms and types are defined simultaneously as one syntactic category, called pseudo-expressions. The typing rules define then the notion of derivability $\Gamma \vdash A : B$ and implicitly, in this way, also the notion of well-defined type and the notion of well-typed term.

For the definition of the systems of the λ-cube, their properties, related terminology and notation, see the preliminary section 1.2.3 and the references given there. Here, we present the systems of the λ-cube as stratified systems of expressions, where β-reduction is restricted to particular levels of expressions. This restriction of β-reduction will later be used in the formalisation of context-related rewriting in our $\lambda[\,]$-cube. Furthermore, we look at meta-contexts of the λ-cube, which are not defined in [Bar92]. We argue that meta-contexts cannot be defined in

[1] It is important to note that we consider here the version of the simply typed lambda calculus $\lambda^{\to}$ à la Church presented in Section 1.2.2. Another version of $\lambda^{\to}$ à la Church is the system $\lambda^{\to}$ of the λ-cube as defined in Section 1.2.3. The two versions of $\lambda^{\to}$ are essentially the same, due to the dependencies that are allowed for $\lambda^{\to}$ of the λ-cube. In particular, based on the allowed dependencies, terms and types of the λ-cube system $\lambda^{\to}$ are two different syntactic categories (cf. Lemma 5.1.14 in [Bar92]).

an informal way as in the untyped and simply typed lambda calculus. For the definition of meta-contexts, communication needs to be formalised. Therefore, we will not define meta-contexts for the λ-cube here directly. Meta-contexts of the λ-cube will be defined later through the $\lambda[\,]$-cube, where context-related issues, including communication, are formalised.

Before going into details of the λ-cube, we discuss the possible dependencies between terms and types in general.

Dependencies between terms and types. We list the four possible sorts of dependencies. This informal discussion is based on Barendregt's explanation in [Bar92], Section 5.1.

i) Terms depending on terms: Consider the $\lambda^{\to}$-expression $\lambda x : \alpha.\, f\,x\,x$. This expression is an example of a term (i.e. $f\,x\,x$) depending on terms (denoted by the term variable x in $f\,x\,x$). This sort of dependency is a natural one by the formation of λ-terms, which formalise functions as abstractions over terms.

ii) Terms depending on types: Consider the $\lambda 2$-expression $\Lambda\alpha.\, \lambda x : \alpha.\, x$. It is an example of a term (here, $\lambda x : \alpha.\, x$) depending on types (denoted by the type variable α). In $\lambda 2$, the type of this expression is $\forall\alpha.\, \alpha{\to}\alpha$. Such types are called polymorphic, and this sort of dependency is said to capture λ-terms with polymorphic types.

iii) Types depending on types: Consider the $\lambda\underline{\omega}$-expression $\lambda\alpha.\, \alpha{\to}\alpha{\to}\alpha$. It is an example of a type (i.e. $\alpha{\to}\alpha{\to}\alpha$) depending on types (denoted by the type variable α). Alternatively, the expression $\lambda\alpha.\, \alpha{\to}\alpha{\to}\alpha$ can be understood as a function from types to types. Hence, it is said to be an element of, in informal notation, *type*$\to$*type*, that is, $(\lambda\alpha.\, \alpha{\to}\alpha{\to}\alpha) : (\mathit{type}{\to}\mathit{type})$. The expressions like *type*$\to$*type* are in $\lambda\underline{\omega}$ called *kinds*. The elements of kinds describe how new types can be formed from other types *within* the syntax of types: the expression $\lambda\alpha.\, \alpha{\to}\alpha{\to}\alpha$ says that, if τ is a type then $\tau{\to}\tau{\to}\tau$ is a type too.

iv) Types depending on terms: Consider the λP-expression $\lambda x : \tau.\, \sigma$ where τ and σ are types and where x may occur in σ. This expression is an example of a type (i.e. σ) depending on terms (denoted by the term variable x in σ). This expression describes how a new type can be formed using terms *within* the syntax of terms and types: if M is a term then $\sigma[\![x := M]\!]$ is a type. Types depending on terms are traditionally called dependent types. This sort of dependency allows, for example, predicates and cartesian products.

Analogously to the expression $\lambda\alpha.\, \alpha{\to}\alpha{\to}\alpha$ being considered as a function from types to types, the expression $\lambda x : \tau.\, \sigma$ can be considered as a function from terms to types. In the λ-cube system λC, such expressions are also said to be elements of kinds.

The permission to λ-abstract in these expressions is crucial. For example, the term $\lambda x : \alpha.\, x$ and the type $\alpha{\to}\alpha$ can be seen to depend on the type variable α

in $\lambda^{\to}$, but in this system abstraction over types is not allowed. Therefore, in $\lambda^{\to}$ terms and types do not depend on types.

The first three sorts of dependencies still allow types and terms to be defined separately: namely, first types and then terms. However, the last sort of dependency introduces terms in types, and consequently, terms and types become the same syntactic category.

Furthermore, by allowing the last two sorts of dependencies, β-reduction is introduced on elements of kinds too. For example,

$$\begin{array}{rcl}(\lambda\alpha.\,\alpha{\to}\alpha{\to}\alpha)\tau & \to_\beta & \tau{\to}\tau{\to}\tau \\ (\lambda x:\tau.\,\sigma)\,M & \to_\beta & \sigma\,[\![x:=M]\!]\,.\end{array}$$

Expressions. Here we address three subjects: (i) implementation of the control over dependencies in the λ-cube, (ii) a slightly adapted terminology of the λ-cube, and (iii) the levels in which legal expressions are arranged by the typing relation.

First, we repeat the definition of the pseudo-expressions of the λ-cube (cf. Definition 1.2.59):

$$A ::= \; x \;\mid\; s \;\mid\; (AB) \;\mid\; (\lambda x:A.\,B) \;\mid\; (\Pi x:A.\,B)$$

where $s \in \mathcal{S} = \{*, \Box\}$. Recall also that a pseudo-expression A is called a legal expression if there are Γ and B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.

The dependencies described above can be characterised by the four cases according to what the abstracted variable (i.e. x in $\lambda x.\,A$) stands for and to what the body of the abstraction (i.e. A in $\lambda x.\,A$) is. The first two dependencies describe the ways in which new terms may be formed: in both cases the body is a term. The last two cases describe the ways in which new types can be formed: the body of the abstraction is a type.

For the purpose of expressing the dependencies and governing the expression formation accordingly, in the λ-cube two sorts are introduced: $*$ and $\Box$. The sort $*$ contains types, i.e. $\tau : *$ amounts to saying that τ is a type. Types will contain terms depending on terms and terms depending on types. The sort $\Box$ contains *kinds*, i.e. $\kappa : \Box$ amounts to saying that κ is a kind. A dependency is then expressed by a pair over $\{*, \Box\}$. The pair (s_1, s_2) denotes the dependency of the elements of sort s_2 on the elements of sort s_1. For example, the pair $(*, \Box)$ denotes that elements of kinds may depend on elements of types. Each system of the λ-cube is parametrised by a set $\mathcal{R}$, expressing which dependencies are allowed in that particular system.

The formation of types and kinds is governed by the specific typing rules, which are parametrised by $\mathcal{R}$. Each dependency pair (s_1, s_2) in $\mathcal{R}$ indicates a particular specific typing rule:

$$(Abs)\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\Pi x:A.\,B) : s_2} \qquad \text{if } (s_1, s_2) \in \mathcal{R}.$$

In this rule, s_1 determines the sort of the abstracted variable x, s_2 determines the sort of the body of the abstraction. In the λ-cube, the thus formed object (i.e. a type or a kind) $\Pi x:A.\,B$ is by default[2] of sort s_2.

[2] In contrast, in PTSs the newly formed object may be of some other sort.

In the λ-cube, the attention is focused on legal expressions, that is, on the pseudo-expressions which are 'well-typed' with respect to the typing rules of a particular system. With terms and types defined simultaneously (see above), only by imposing typing restrictions do we obtain well-defined types and kinds and their well-typed elements.

The terminology of the λ-cube regarding expressions is refined with respect to the terminology of the lambda calculus, due to new sorts of expressions. We adopt here the slightly adapted terminology of the λ-cube system λC (cf. Definition 5.2.3 in [Bar92]), because λC contains all possible dependencies and hence, all sorts of (legal) expressions. The terminology of λC is based on the fact that legal expressions can be seen as stratified into five levels[3]: the level of sorts, the level of kinds, the level of types, the level of elements of kinds and the level of elements of types:

element of kind : kind : $\Box$

element of type : type : $*$.

The examples of dependencies treated above, now in the notation and terminology of the λ-cube, are as follows.

element	:	type or kind	:	sort	needed specific typing rule
$\lambda x:\tau.\, f\,x$	:	$\Pi x:\tau.\,\sigma$	:	$*$	$(*,*)$
$\lambda\alpha:*.\,\lambda x:\alpha.\,x$	:	$\Pi\alpha:*.\,\sigma$	:	$*$	$(\Box,*)$
$\lambda\alpha:*.\,\alpha\to\alpha\to\alpha$	:	$\Pi\alpha:*.\,*$	:	$\Box$	$(\Box,\Box)$
$\lambda x:\tau.\,\sigma$	:	$\Pi x:\tau.\,*$	:	$\Box$	$(*,\Box)$

We discuss the structure of legal expressions in more detail. An important observation related to expressions and rewriting in the λ-cube is that although, next to the standard lambda calculus abstraction λ, a new abstraction Π is introduced, there is no corresponding application. Accordingly, in the conclusion of the typing rule (app), the Π-abstraction is immediately eliminated[4]:

$$(app)\quad \frac{\Gamma \vdash F:(\Pi x:A.\,B) \qquad \Gamma \vdash a:A}{\Gamma \vdash Fa:(B[\![x:=A]\!]).}$$

[3]These levels are however not disjoint, because of the axiom $*:\Box$. For example, if $A:*$ and $a:A$, then

$$\begin{array}{lllll} x & : & A & : * & : \Box \\ x & : & (\lambda y:A.\,A)\,a & : * & : \Box \\ x & : & (\lambda Y:*.\,Y)\,A & : * & : \Box. \end{array}$$

Here, each of the expressions A, $(\lambda y:A.\,A)\,a$ and $(\lambda Y:*.\,Y)\,A$ is both an element and a type. The expression $*$ is both a kind and a sort.

[4]This is in contrast to the practice in the Automath languages (see [Daa73, Ned73]), where there is only one abstraction and only one application, both of which may occur in a type. See also [KBN99] where both β-reduction and Π-reduction are used in the λ-cube.

This means that a functional type like $\Pi x : A.\, B$ may be formed, but if an argument a is provided, the functionality of the type is adapted *and computed* immediately: $B[\![x := A]\!]$. There is no explicit postponement of such a computation, and hence, no corresponding application–abstraction elimination rule.

Some examples of legal expressions per level and a short analysis of their structure are given in the preliminary section 1.2.3.

β-reduction. In the λ-cube, β-reduction is defined on pseudo-expressions. However, on *legal* expressions, β-reduction can be seen as being restricted to the level of elements (of types or kinds), because β-redexes are elements: Consider the redex $(\lambda x : A.\, b)\, a$ and let it be a legal expression. Then, by using the Generation and Substitution lemmas 1.2.71 and 1.2.72 for the λ-cube, there exist Γ, B and s with

$$\begin{array}{lll} \Gamma & \vdash & (\lambda x : A.\, b) \;:\; (\Pi x : A.\, B) \;:\; s, \\ \Gamma & \vdash & a \;:\; A, \qquad\qquad\qquad\qquad \text{and} \\ \Gamma & \vdash & (\lambda x : A.\, b)\, a \;:\; B[\![x := a]\!] \;:\; s. \end{array}$$

Hence, the redex $(\lambda x : A.\, b)\, a$ is an element.

With types possibly depending on elements, redexes may *occur* in types too. Again, the positions of redexes in a type or kind (and, in general, in any expression) are restricted by the fact that redexes are elements. For example, redexes cannot occur on the spine of a kind: $\Gamma \not\vdash (\lambda x : A.\, \Pi \vec{y} : \vec{B}.\, *)\, a \;:\; \Box$.

Note here that, in addition to α-conversion, β-reduction is the only rewrite relation in the λ-cube. As we have already remarked, although a new abstraction, namely Π, is introduced, there is no corresponding application, and hence, no redexes of the form $(\Pi x : A.\, \_) \cdot a$ where $\cdot$ denotes the application related to Π.

Meta-contexts. For the definition of the λ-cube and investigation of the properties of the λ-cube, meta-contexts are not needed. Accordingly, in [Bar92] there is no definition of meta-contexts.

However, an attempt to give a definition of meta-contexts using informal notation as in the lambda calculus, fails. We claim here that, in order to define meta-contexts in the λ-cube in a way analogous to the definition of meta-contexts in the untyped lambda calculus, one needs a better representation of holes together with communication. We set out the assumptions regarding meta-contexts that seem natural, and describe the problem. The assumptions are the following.

i) In the λ-cube the attention is focused on legal expressions. Then it seems natural to define (legal) meta-contexts, informally, as legal expressions with holes. In the sequel, in the informal notation holes will be denoted by $[]^A$ where A is the type or kind of the hole.

ii) The notion of legal expression is defined by the typing rules. Then, with meta-contexts defined as legal expressions with holes, it seems natural to define the notion of meta-context by using the same typing rules. In the typing derivations, it seems natural to treat holes as constants, that is, as free variables which are not allowed to be abstracted.

iii) In general, hole filling is an integral part of the definition of meta-contexts (see Section 2.1). As in the case of the untyped lambda calculus, hole filling is, informally, a literal replacement of a hole by a legal expression of the same type or kind as the hole.

With these assumptions, meta-contexts and hole filling are not well-defined. The problem lies in the fact that the typing rules employ substitution and β-conversion, and as we have shown in Section 2.2, communication of a hole is not preserved under these transformations. This is illustrated by the example given below.

Example 6.1.1 Consider the typing statement

$$Y : * \vdash (\lambda X : *. \lambda x : []^*. x) Y : (\Pi x : []^*. []^*)$$

and a fragment of its derivation:

$$\frac{Y : * \vdash (\lambda X : *. \lambda x : []^*. x) : (\Pi X : *. \Pi x : []^*. []^*) \qquad Y : * \vdash Y : *}{Y : * \vdash (\lambda X : *. \lambda x : []^*. x) Y : (\Pi x : []^*. []^*).}$$

The hole of the element and the hole of the type originate from the same hole: the type of x. Therefore, they should be considered as occurrences of the same hole, and accordingly, filled by the same expression. However, while filling the hole of the element and the hole of the type by X where $X : *$ is 'legal' in the assumption, it results in an incorrect typing statement in the conclusion: $(\lambda X : *. \lambda x : X. x) : (\Pi X : *. \Pi x : X. X)$ but not $(\lambda X : *. \lambda x : X. x) Y : (\Pi x : X. X)$. The problem arises because in this derivation step, the information $[\![X := Y]\!]$ is not recorded by the hole in the type.

Hence, in order to define meta-contexts, a notation for holes with explicitly denoted communication is necessary. Furthermore, as in the case of lambda calculus (see Section 2.2), one can show that substitution, α-conversion and β-reduction do not 'commute' with filling of holes.

6.2 Introduction to the context cube

We describe here the $\lambda[\,]$-cube informally; the definition is given in the next section. This section extends the description of our approach to formalisation of contexts, which was given in Section 2.5.

A collection of context calculi related to the λ-cube. The $\lambda[\,]$-cube defines a collection of eight context calculi. These context calculi extend the eight systems of the λ-cube with contexts. The $\lambda[\,]$-cube provides an adequate notation for meta-contexts of the λ-cube, where β-reduction on contexts is gained for free.

Context calculi. On (representation of) contexts, also the standard relations and operations are defined: typing, substitution, α-conversion, and β-reduction. Furthermore, the systems of the $\lambda[\,]$-cube include variables over (representations of) contexts and functions over (representation of) contexts.

We stress that context-related rewriting in the $\lambda[\,]$-cube is defined on the same level where β-rewriting is allowed in the λ-cube, namely on the level of elements. This is the level where the problem of commutation between β-reduction and hole filling occurred in the informal notation. Incidentally, the consequence of this choice is that we stay within the framework of the λ-cube.

The notion of context. Indirectly, the $\lambda[\,]$-cube defines a notion of context for the λ-cube. It formalises contexts with many holes, which in turn may occur an arbitrary number of times. A hole may occur at any position where otherwise a variable may occur. The holes of a context are filled sequentially.

Typing. The typing relation in each system of the $\lambda[\,]$-cube makes the assumptions listed above. Moreover, the typing relation will satisfy the following: if C is a context of type or kind T, then $C[\vec{a}]$ is of type or kind $T[\vec{a}]$ for suitable expressions $\vec{a}$. This restriction filters out typing statements like $(\lambda x : []^*.\, x) : (\Pi x : A.\, A)$, where the typing relation assumes more information about filling of holes than available in the expression and its type or kind.

The typing relation in each system of the $\lambda[\,]$-cube is defined in such a way that the levels are preserved: if a meta-context of the λ-cube is on the level of elements, types or kinds, then its representation within the $\lambda[\,]$-cube is at the same level.

Properties. Technically, we arrive at a very simple extension of the λ-cube. The syntax of the expressions is extended with new abstractions and applications related to representing contexts. These new expression constructors can be understood as macros over the syntax of the λ-cube. In particular, each system of the $\lambda[\,]$-cube can be translated into the corresponding system of the λ-cube. That means that the whole $\lambda[\,]$-cube can be encoded into the λ-cube.

The $\lambda[\,]$-cube is rather expressive. For example, as we will show later, 'polymorphic' de Bruijn's segments with dependent types can be represented within the $\lambda[\,]$-cube.

Representing the lambda cube with contexts. We give the intuition behind the representation method. The representation method builds upon the method described in Section 2.5 for the context calculus λc; see also Example 6.3.9.

- Legal expressions of the λ-cube: These expressions can be represented directly in the $\lambda[\,]$-cube, because each system $\lambda[S]$ of the $\lambda[\,]$-cube is an extension of the corresponding system λS of the λ-cube.

- Holes: Holes are represented by hole variables $h, g, H, \ldots$ applied to a sequence of n expressions $\vec{N}$ that keep track of the relevant α, β-changes in the context. So, holes are represented as $h\langle\vec{N}\rangle_n$, where $\_\langle\_\rangle_n$ denotes multiple application.

- Communication: In addition to multiple application, which is used in the representation of communication around the holes, we introduce the multiple abstractions Λ and $\underline{\Lambda}$ for representing the expressions that are to be put into holes and their types or kinds. In general, if M is an expression that is intended to be put into a hole and where $\vec{x}$ are intended to become bound upon filling,

and if T is the type (or kind) of M, then M and T are represented in the $\lambda[\,]$-cube as $\Lambda_n\vec{x}{:}\vec{A}.\,M$ and $\underline{\Lambda}_n\vec{x}{:}\vec{A}.\,T$ respectively, and the intention is that $\Lambda_n\vec{x}{:}\vec{A}.\,M$ is of type $\underline{\Lambda}_n\vec{x}{:}\vec{A}.\,T$. Upon filling the expression into a hole $h\langle\vec{N}\rangle_n$, that is, upon replacing the hole variable by the expression, commutation can be computed by a multiple β-rewrite step:

$$(\Lambda_n\vec{x}{:}\vec{A}.\,M)\langle\vec{N}\rangle_n \to M\,[\![\vec{x} := \vec{N}]\!]\,.$$

If $\underline{\Lambda}_n\vec{x}{:}\vec{A}.\,T$ is the type of $\Lambda_n\vec{x}{:}\vec{A}.\,M$, then the type of $(\Lambda_n\vec{x}{:}\vec{A}.\,M)\langle\vec{N}\rangle_n$ is $T[\![\vec{x} := \vec{N}]\!]$.

- Contexts, hole filling: Contexts are considered as functions over the possible contents of its holes; that is, contexts are functions over hole variables. The abstractor for the hole variables is denoted by δ. Hole variables of a context will be abstracted sequentially. In general, if C is a context of type (or kind) T_C with n holes (holes are represented as above), then C is represented as $\delta h_1{:}T_1.\ldots\delta h_n{:}T_n.\,C$ and its type is represented as $\underline{\delta}h_1{:}T_1.\ldots\underline{\delta}h_n{:}T_n.\,T_C$. In general, in (the representation of) types and kinds, hole variables are abstracted by $\underline{\delta}$. The intention is again that $\delta h_1{:}T_1.\ldots\delta h_n{:}T_n.\,C$ is of type $\underline{\delta}h_1{:}T_1.\ldots\underline{\delta}h_n{:}T_n.\,T_C$.

  Hole filling is then represented sequentially as follows. If $\delta h{:}T.\,U$ is a context and if $\Lambda_n\vec{x}{:}\vec{A}.\,M$ is a communicating expression, then hole filling is represented as $(\delta h{:}T.\,U)\lceil\Lambda_n\vec{x}{:}\vec{A}.\,M\rceil$ and computed by a version of the β-rewrite rule:

$$(\delta h{:}T.\,U)\lceil\Lambda_n\vec{x}{:}\vec{A}.\,M\rceil \to U\,[\![h := \Lambda_n\vec{x}{:}\vec{A}.\,M]\!]\,.$$

  If $\underline{\delta}h{:}T.\,T_U$ is the type of $\delta h{:}T.\,U$, then the type of $(\delta h{:}T.\,U)\lceil\Lambda_n\vec{x}{:}\vec{A}.\,M\rceil$ is $T_U[\![h := \Lambda_n\vec{x}{:}\vec{A}.\,M]\!]$.

  Composition can be encoded within the $\lambda[\,]$-cube.

- Sorts: Using the constructors of the $\lambda[\,]$-cube, we will represent terms and types, as in the λ-cube, and moreover, contexts and communicating terms. Contexts will not be considered as a new sort of expressions, because contexts are terms or types with holes. Hence, the dependencies between contexts, terms and types can be expressed by $*$ and $\Box$ in a way that is analogous to controlling the dependencies in the λ-cube.

  However, communicating expressions will be considered as special expressions, because such expressions are combined with contexts, terms and types in a different way. For example, a context may depend on a communicating expression, but a communicating expression may only depend on terms and types. For denoting the sorts of communicating expressions two auxiliary sorts are introduced: $\triangle_*$, which denotes communicating types, and $\triangle_\Box$, which denotes communicating kinds.

Remark 6.2.1 The four constructors Λ_n, $\langle\,\rangle_n$, δ, and $\lceil\,\rceil$ are also constructors of the context calculus λc; their role in λc is the same. The two abstractors $\underline{\Lambda}_n$ and $\underline{\delta}$ are new also with respect to λc.

In sum, the elements are treated as in the context calculus λc; the abstractors $\underline{\Lambda}$ and $\underline{\delta}$ are introduced for representing the types and kinds (with holes). Holes in types and kinds are filled by substitution, filling of holes in elements can be denoted and its computation can be postponed. This explanation will also be visible in the typing rules.

6.3 Definition of the context cube

This section presents the definition of expressions, rewriting and typing of the eight systems of the context cube.

Let $\mathcal{S} = \{*, \Box, \triangle_*, \triangle_\Box\}$ be the set of sorts. Let $\mathcal{V}$ be a countably infinite set of variables.

Definition 6.3.1 (Pseudo-expressions) Let $s \in \mathcal{S}$ and $x, \vec{x} \in \mathcal{V}$. The set of pseudo-expressions A of the $\lambda[\,]$-cube is defined inductively by

$$\begin{array}{rcrrrr} A & ::= & s \mid & x \mid & (\Pi x : A.\, A) \mid & (\lambda x : A.\, A) \mid & (A\,A) \mid \\ & & & & (\underline{\Lambda}_n \vec{x} : \vec{A}.\, A) \mid & (\Lambda_n \vec{x} : \vec{A}.\, A) \mid & (A \langle \vec{A} \rangle_n) \mid \\ & & & & (\underline{\delta} x : A.\, A) \mid & (\delta x : A.\, A) \mid & (A \lceil A \rceil) \end{array}$$

where $|\vec{x}| = |\vec{A}| = n$.

Notation. As usual, standard abbreviations regarding brackets apply, including the association to the left. Also, standard abbreviations for sequences of n pseudo-expressions as $\vec{A}$ and sequences of n variables as $\vec{x}$ apply. When convenient, consecutive abstractions $(\lambda x_1 : A_1. \ldots (\lambda x_n : A_n.\, B))$ and $(\Pi x_1 : A_n. \ldots (\Pi x_n : A_n.\, B))$ will be abbreviated by $\lambda \vec{x} : \vec{A}.\, B$ and $\Pi \vec{x} : \vec{A}.\, B$, respectively, and moreover, if $A_1 \equiv \ldots \equiv A_n$, then we write $\lambda \vec{x} : A.\, B$ and $\Pi \vec{x} : A.\, B$, respectively. In the multiple abstractions Λ_n and $\underline{\Lambda}_n$, if $n = 0$ then we write $\Lambda \epsilon.\, A$ and $\underline{\Lambda} \epsilon.\, A$. Furthermore, we will omit the index n of Λ, $\underline{\Lambda}$, and $\langle\,\rangle$, and assume that the arities of these constructors and the number of their arguments match. As in the case of lambda calculus, if x does not occur free in B, then $\Pi x : A.\, B = A \to B$. Arbitrary expressions will be denoted by A,B,C,.... Hole variables will be denoted by $h, g, k, h_1, H, G \ldots$; arbitrary variables will be denoted by $x, y, x', x_1, X, Y, \ldots$.

Substitution over pseudo-expressions is defined in a standard way; we will skip the definition here.

On pseudo-expressions, rewrite relations are defined via rewrite rules. The rewrite rules are split into the standard β-reduction and the context-related rewriting. The context-related rewriting comprises communication and hole filling.

Definition 6.3.2 (Rewrite rules) The rewrite relations of the $\lambda[\,]$-cube are induced by two collections of rewrite rules, the lambda calculus rewrite rule and the context rewrite rules. The two collections of rewrite rules are given below.

i) The lambda calculus rewrite rule is:

$$(\lambda x : A.\, B)C \;\to\; B[\![x := C]\!] \qquad (\beta)$$

ii) The context rewrite rules are:

$$(\Lambda \vec{x} : \vec{A}.\, B)\langle \vec{C} \rangle \;\to\; B[\![\vec{x} := \vec{C}]\!] \qquad (\textcircled{m}\beta)$$
$$(\delta h : A.\, B)\lceil C \rceil \;\to\; B[\![h := C]\!]. \qquad (\mathit{fill})$$

As usual, it is assumed that the bound variables in the rewrite rules are renamed to avoid variable capturing.

Notation. Let $\to_c$ denote the rewrite relation generated by the context rewrite rules of the $\lambda[\,]$-cube.

The context rewrite rules can be understood as instances of the β-rewrite rule, because they too are application–abstraction elimination rules. The rule $(\textcircled{m}\beta)$ is a multiple version of the β-rule and the rule (*fill*) is a version of the β-rule dealing with an abstraction over a hole variable.

Note that there is no composition in either the syntax of the pseudo-expressions or the rewrite rules. We will show later that composition can be represented within the systems of the $\lambda[\,]$-cube.

Pseudo-expressions and the rewrite rules form a rewrite system. We will call it the underlying calculus of the context cube, or the $\lambda[\,]$-calculus for short, and use it only as an auxiliary system.

Definition 6.3.3 (The underlying calculus: the $\lambda[\,]$-calculus)
The underlying calculus of the context cube, called the $\lambda[\,]$-calculus, consists of the set of pseudo-expressions as defined in Definition 6.3.1 and the rewrite relations as defined in Definition 6.3.2.

The systems of the context cube are presented in a uniform way, analogously to the definition of λ-cube (see [Bar92]). The eight systems of the $\lambda[\,]$-cube are $\lambda[\to]$, $\lambda[2]$, $\lambda[P]$, $\lambda[P2]$, $\lambda[\underline{\omega}]$, $\lambda[\omega]$, $\lambda[P\underline{\omega}]$, and $\lambda[C]$. The names for these systems are derived from the names of the systems of the λ-cube. As we will show later, system $\lambda[S]$ of the $\lambda[\,]$-cube corresponds to the system λS of the λ-cube with contexts.

Each of the eight systems is defined on legal expressions with respect to a particular collection of the typing rules. The typing rules are divided into the general typing rules and the specific typing rules. The general typing rules are the rules that are present in each of the eight systems; these rules guide the formation of elements, allow for conversion in their types and kinds, and guide the formation of legal bases. The specific typing rules control the formation of types and kinds

according to the allowed dependencies between terms and types[5] in a particular system of the $\lambda[\,]$-cube. The dependencies that are allowed in a particular system are expressed by $\mathcal{R} \subseteq \{*, \Box\} \times \{*, \Box\}$ and each system includes a different subset of the specific typing rules. Hence, the specific rules are parametrised by $\mathcal{R}$.

The eight systems of the $\lambda[\,]$-cube and the dependencies $\mathcal{R}$ allowed in a particular system are displayed in Table 6.1. The dependencies $\mathcal{R}$ of a system $\lambda[S]$ are the same as the dependencies of the system λS of the λ-cube.

system $\lambda[S]$	parametrised by $\mathcal{R}$			
$\lambda[\rightarrow]$	$(*,*)$			
$\lambda[2]$	$(*,*)$	$(\Box,*)$		
$\lambda[P]$	$(*,*)$		$(*,\Box)$	
$\lambda[P2]$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda[\underline{\omega}]$	$(*,*)$			$(\Box,\Box)$
$\lambda[\omega]$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda[P\underline{\omega}]$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda[C]$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

Table 6.1: The systems of the $\lambda[\,]$-cube

Statements, declarations and pseudo-bases are defined analogously to the corresponding notion in the λ-cube.

Definition 6.3.4

i) A statement is of the form $A : B$ with $A, B \in \mathcal{T}$.

ii) A declaration is a statement of the form $x : A$ where $x \in \mathcal{V}$.

iii) A pseudo-basis is a finite, ordered sequence of declarations with distinct variables. The empty basis is denoted by $\emptyset$.

So, a pseudo-basis is an *ordered* sequence of declarations. This is comparable to the case of the λ-cube, but contrary to the case of for example simply typed lambda calculus $\lambda^{\rightarrow}$ or the applications of the context calculus λc. The ordering is a consequence of the fact that if $x_1 : A_1, \ldots, x_n : A_n$ is a pseudo-basis, then A_j may depend on x_i for i and j with $1 \leq i < j \leq n$.

The typing relation employs only one basis, viz. $\Gamma \vdash A : B$. This basis contains declarations of both term variables and hole variables. The reason for using only one basis in the $\lambda[\,]$-cube is the fact that the type of a hole variable may depend on a term variable, and vice versa.

[5]Here we mean, terms and types in the terminology of, for example, the simply typed lambda calculus.

Definition 6.3.5 (Typing) Let $\lambda[S]$ be a system of the $\lambda[\,]$-cube, parametrised by $\mathcal{R}$. Let A and B be pseudo-expressions and let Γ be a pseudo-basis. Then $A : B$ can be derived from the pseudo-basis Γ, denoted by $\Gamma \vdash_{\lambda[S]} A : B$, if $\Gamma \vdash A : B$ can be derived using $\mathcal{R}$ and the typing rules displayed in Figures 6.1 and 6.2.

The systems of the $\lambda[\,]$-cube are defined on legal expressions provided with β-reduction and the context-related rewrite relations.

Definition 6.3.6 (The $\lambda[\,]$-cube) Each of the eight systems of the $\lambda[\,]$-cube are defined as follows.

Let $\lambda[S]$ be one of the systems of the $\lambda[\,]$-cube listed in Table 6.1. Let $\lambda[S]$ be parametrised by $\mathcal{R}$. Then, the system $\lambda[S]$ consists of the legal expressions as defined by the typing rules of Figures 6.1 and 6.2 with respect to $\mathcal{R}$, and of the rewrite relations of Definition 6.3.2.

We first establish some notation and terminology, and then comment on the typing system.

Notation. If the system is irrelevant, the index $\lambda[S]$ in $\Gamma \vdash_{\lambda[S]} A : B$ is omitted. Furthermore, $\Gamma \vdash A : B : C$ is a shorthand for the two derivations $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$.

Definition 6.3.7 Let A be a pseudo-expression.

i) A typing statement is of the form $\Gamma \vdash A : B$.

ii) Let Γ be a pseudo-basis. Then, Γ is called a (legal) basis if $\exists P, Q \in \mathcal{T}\ (\Gamma \vdash P : Q)$.

iii) A is called a (legal) expression if $\exists \Gamma, B\ (\Gamma \vdash A : B$ or $\Gamma \vdash B : A)$.

iv) A is called a kind if there is a basis Γ such that $\Gamma \vdash A : \Box$.

v) A is called a communicating kind if there is a basis Γ such that $\Gamma \vdash A : \triangle_{\Box}$.

vi) A is called a type if there is a basis Γ such that $\Gamma \vdash A : *$.

vii) A is called a communicating type if there is a basis Γ such that $\Gamma \vdash A : \triangle_{*}$.

viii) A is called an element if $\exists B \in \mathcal{T} \exists s \in \mathcal{S}\ (\Gamma \vdash A : B : s)$.

We comment on the typing system. The typing rules of the $\lambda[\,]$-cube, displayed in Figures 6.1 and 6.2, include the typing rules of the λ-cube. These are the general rules (*axiom*), (*var*), (*weak*), (*conv*), (*abs*) and (*app*), and the specific rule (Π). Here, in the $\lambda[\,]$-cube, these rules are of course adapted for dealing also with new sorts of expressions: the sort s in (*var*), (*weak*) and (*conv*) may also be $\triangle_{*}$ or $\triangle_{\Box}$; by the rules (*var*) and (*weak*) both term variables and hole variables may be

introduced; and in the conversion $B =_{\beta c} B'$ of the rule (*conv*) also the context-related rewriting may be employed.

In addition to the typing rules of the λ-cube, the $\lambda[\,]$-cube contains new typing rules which deal with formation of new sorts of expressions. The additional rules are the general rules (*mabs*), (*mapp*), (*habs*), and (*fill*), and the specific rules ($\underline{\Lambda}$) and ($\underline{\delta}$). The rules (*mabs*) and (*mapp*) deal with communication: they guard the typing of communication around holes and the typing of elements to be put into holes. The rules (*habs*) and (*fill*) deal with formation of contexts and filling of holes of a context.

By the rules (*mabs*) and (*habs*), the abstraction of variables is reflected in both the element and its type or kind: in the case of (*habs*), if $\Gamma, h : A \vdash b : B$ then $\Gamma \vdash (\delta h : A.\, b) : (\underline{\delta} h : A.\, B)$. This is comparable to typing of λ-abstractions by the rule (*abs*). By the rules (*mapp*) and (*fill*), the 'application' is postponed in the element but it is computed immediately in its type or kind: in the case of (*fill*), for example, if $\Gamma \vdash a : A$ and $\Gamma \vdash (\delta h : A.\, b) : (\underline{\delta} h : A.\, B)$ then $\Gamma \vdash (\delta h : A.\, b)\lceil a \rceil : B[\![x := a]\!]$. This is comparable to typing of applications by the rule (*app*). By these rules a (β-, $\textcircled{m}\beta$- or *fill*-)redex is formed only in the element. Hence, in general, although a redex may also occur in a type of a kind, the redex itself is an element.

The specific rules ($\underline{\Lambda}$) and ($\underline{\delta}$) govern the formation of communicating types and communicating kinds, and the formation of the types and kinds of contexts, respectively. They are parametrised by the allowed dependencies in a particular system, which are expressed by $\mathcal{R}$. The requirement in the rule ($\underline{\Lambda}$) in case $|\vec{x}| = 0$, that there is an s' with $(s', s) \in \mathcal{R}$ ensures that no abstractions are allowed over expressions over which otherwise no abstractions are allowed. This condition excludes expressions like $\Lambda\epsilon.\, A$ and $\underline{\Lambda}\epsilon.\, *$ with $A : *$, if otherwise also $\Lambda x : B.\, A$ and $\underline{\Lambda} x : B.\, *$ are not allowed for any B (i.e. if there is no $(s, \Box)$ in $\mathcal{R}$).

By (thoroughly) inspecting the typing rules of the $\lambda[\,]$-cube, one may see that, as in the case of the λ-cube, legal expressions are stratified into levels. Because of the new sorts in the $\lambda[\,]$-cube, its level structure is more involved than in the λ-cube:

element : type : $*$	element : communicating type : $\triangle_*$
element : kind : $\Box$	element : communicating kind : $\triangle_\Box$.

Here, as in the case of the λ-cube, the levels of elements on one hand, and the levels of types and kinds on the other hand overlap.

We give two examples dealing with legal expressions. The aim of the first example is to give a flavour of the levels into which the legal expressions are stratified.

The second example illustrates the expressive power of the $\lambda[\,]$-cube. It treats meta-contexts of the λ-cube: the examples considered can be understood as translations of meta-contexts. Furthermore, the second example shows that the $\lambda[\,]$-cube has a greater expressivity than just representing meta-contexts. In the $\lambda[\,]$-cube also functions over (representations of) contexts can be represented, as well as intermediate results of context-related computation. At the end of this chapter, we

$$(axiom) \quad \emptyset \vdash * : \Box$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad s = *, \Box, \triangle_*, \triangle_\Box$$

$$(weak) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad s = *, \Box, \triangle_*, \triangle_\Box$$

$$(conv) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \quad B =_{\beta c} B'}{\Gamma \vdash A : B'} \quad s = *, \Box, \triangle_*, \triangle_\Box$$

$$(abs) \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\, B) : s}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi x : A.\, B)} \quad s = *, \Box$$

$$(app) \quad \frac{\Gamma \vdash F : (\Pi x : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : (B[\![x := a]\!])}$$

$$(mabs) \quad \frac{\Gamma, \vec{x} : \vec{A} \vdash b : B \qquad \Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}.\, B) : \triangle_s}{\Gamma \vdash (\Lambda \vec{x} : \vec{A}.\, b) : (\underline{\Lambda}\vec{x} : \vec{A}.\, B)} \quad s = *, \Box$$

$$(mapp) \quad \frac{\Gamma \vdash F : (\underline{\Lambda}\vec{x} : \vec{A}.\, B) \qquad \Gamma \vdash a_i : A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!]}{\Gamma \vdash F\langle \vec{a} \rangle : (B[\![\vec{x} := \vec{a}]\!])} \quad \text{for } 1 \leq i \leq |\vec{a}| = |\vec{A}|$$

$$(habs) \quad \frac{\Gamma, h : A \vdash b : B \qquad \Gamma \vdash (\underline{\delta} h : A.\, B) : s}{\Gamma \vdash (\delta h : A.\, b) : (\underline{\delta} h : A.\, B)} \quad s = *, \Box$$

$$(fill) \quad \frac{\Gamma \vdash F : (\underline{\delta} h : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\lceil a \rceil : B[\![h := a]\!]}$$

Figure 6.1: General part of the type system for the $\lambda[\,]$-cube

$$(\Pi)\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.\, B) : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{R}$$

$$(\underline{\Lambda})\quad \frac{\Gamma, x_1 : A_1, \ldots, x_i : A_i \vdash A_{i+1} : s_{i+1} \quad \Gamma, \vec{x} : \vec{A} \vdash B : s}{\Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}.\, B) : \triangle_s} \quad \begin{array}{l}\text{if } (s_i, s) \in \mathcal{R} \\ \text{for all } 1 \leq i \leq |\vec{x}|, \\ \text{and, in case } |\vec{x}| = 0, \\ \text{if } \exists s' \text{ with } (s', s) \in \mathcal{R}\end{array}$$

$$(\underline{\delta})\quad \frac{\Gamma \vdash A : \triangle_{s_1} \qquad \Gamma, h : A \vdash B : s_2}{\Gamma \vdash (\underline{\delta} h : A.\, B) : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{R}$$

Figure 6.2: System-specific part of the type system for the $\lambda[\,]$-cube

will work out a more involved example dealing with de Bruijn's segments with dependent types and their representation within the $\lambda[\,]$-cube.

Example 6.3.8 We list some examples of typing statements.

i) All legal expressions of the λ-cube are legal expressions of the $\lambda[\,]$-cube, because the $\lambda[\,]$-cube contains the typing rules of the λ-cube (see Lemma 6.4.14). The examples of typing statements of the λ-cube that were given in Section 1.2.3 are also examples of typing statements of the $\lambda[\,]$-cube.

ii) Consider the examples:

$$\begin{array}{rl} X : *, Y : * \vdash_{\lambda[\to]} & (\Lambda x{:}X, y{:}Y.\, x) \;:\; (\underline{\Lambda} x{:}X, y{:}Y.\, X) \;:\; \triangle_* \\ X : *, Y : * \vdash_{\lambda[P]} & (\Lambda x{:}X, y{:}Y.\, X) \;:\; (\underline{\Lambda} x{:}X, y{:}Y.\, *) \;:\; \triangle_\Box \\ X : *, h : (\underline{\Lambda} x{:}X.\, X) \vdash_{\lambda[\to]} & h \;:\; (\underline{\Lambda} x{:}X.\, X) \;:\; \triangle_*. \end{array}$$

In general, the elements of $\triangle_*$ and $\triangle_\Box$, that is, communicating types and communicating kinds respectively, are (always) $\underline{\Lambda}$-abstractions. The reverse is true too: a $\underline{\Lambda}$-abstraction is an element of either $\triangle_*$ or $\triangle_\Box$. Elements of $\underline{\Lambda}$-abstractions are either Λ-abstractions or hole variables. The reverse is true again: a Λ-abstraction or a hole variable is an element of a $\underline{\Lambda}$-abstraction.

iii) Consider the examples and the positions of $h\langle x\rangle$ and $g\langle\rangle$:

$$\begin{array}{rl} X : *, h : (\underline{\Lambda} x : X.\, X) \vdash_{\lambda[\to]} & (\lambda x{:}X.\, h\langle x\rangle) \;:\; (\Pi x{:}X.\, X) \;:\; * \\ X : *, h : (\underline{\Lambda} x : X.\, *) \vdash_{\lambda[P]} & (\lambda x{:}X.\, h\langle x\rangle) \;:\; (\Pi x{:}X.\, *) \;:\; \Box \\ g : (\underline{\Lambda}\epsilon.\, *) \vdash_{\lambda[P]} & (\lambda x{:}\, g\langle\rangle\, .\, x) \;:\; (\Pi x{:}\, g\langle\rangle\, .\, g\langle\rangle) \;:\; * \\ g : (\underline{\Lambda}\epsilon.\, *) \vdash_{\lambda[P]} & (\lambda x{:}\, g\langle\rangle\, .\, g\langle\rangle) \;:\; (\underline{\Lambda} x{:}\, g\langle\rangle\, .\, *) \;:\; \triangle_\Box. \end{array}$$

In general, in meta-contexts of the λ-cube a hole may occur at any position where otherwise a variable may occur. Hence, representations of holes, like

$h\langle x\rangle$ and $g\langle\rangle$, may occur at any position where otherwise a variable (other than a hole variable) may occur.

iv) Consider the examples:

$$\begin{array}{rl}
X:*\vdash_{\lambda[\to]} & (\delta h{:}(\underline{\Lambda}x{:}X.\,X).\,\lambda x{:}X.\,h\langle x\rangle)\\
& \quad :\ (\underline{\delta}h{:}(\underline{\Lambda}x{:}X.\,X).\,\Pi x{:}X.\,X)\ :\ *\\
\emptyset\vdash_{\lambda[P]} & (\delta g{:}(\underline{\Lambda}\epsilon.*).\,\lambda x{:}\,g\langle\rangle\,.\,x)\ :\ (\underline{\delta}g{:}(\underline{\Lambda}\epsilon.*).\,\Pi x{:}\,g\langle\rangle\,.\,g\langle\rangle)\ :\ *\\
X:*,Y:*\vdash_{\lambda[2]} & (\delta h{:}(\underline{\Lambda}x{:}Y.\,Y).\,\lambda x{:}Y.\,h\langle x\rangle)\\
& \quad :\ ((\lambda X{:}*\,.\,\underline{\delta}h{:}(\underline{\Lambda}x{:}X.\,X).\,\Pi x{:}X.\,X)Y)\ :\ *\\
X:*\vdash_{\lambda[2]} & ((\lambda Y{:}*\,.\,\delta h{:}(\underline{\Lambda}x{:}Y.\,Y).\,\lambda x{:}Y.\,h\langle x\rangle)X)\\
& \quad :\ (\underline{\delta}h{:}(\underline{\Lambda}x{:}X.\,X).\,\Pi x{:}X.\,X)\ :\ *.
\end{array}$$

In general, a $\underline{\delta}$-abstraction is an element of either $*$ or $\Box$. A δ-abstraction is an element of a $\underline{\delta}$-abstraction or an element of an expression which is convertible to a $\underline{\delta}$-abstraction. Also, an element of a $\underline{\delta}$-abstraction is either a δ-abstraction or an expressions which is convertible to a δ-abstraction.

v) Consider the examples of multiple application and hole filling:

$$\begin{array}{rl}
X:*,Y:*,a:X,b:Y\vdash_{\lambda[\to]} & (\Lambda x{:}X,y{:}Y.\,x)\langle a,b\rangle\ :\ X\ :\ *\\
X:*,a:X,h:(\underline{\Lambda}x{:}X.\,X)\vdash_{\lambda[\to]} & h\langle a\rangle\ :\ X\ :\ *\\
X:*\vdash_{\lambda[P]} & (\delta g{:}(\underline{\Lambda}\epsilon.*).\,\lambda x{:}\,g\langle\rangle\,.\,x)\lceil\Lambda\epsilon.\,X\rceil\ :\ (\Pi x{:}\,(\Lambda\epsilon.\,X)\langle\rangle\,.*)\ :\ \Box.
\end{array}$$

Note that by the formation of a multiple application or hole filling on the level of elements as above, no multiple application or hole filling is introduced on the level of types or kinds.

Example 6.3.9 We consider some legal expressions of the $\lambda[\,]$-cube, which can be understood as representations of (legal) meta-contexts of the λ-cube. Throughout this example the expressions of the $\lambda[\,]$-cube will be 'translated' into informal, inadequate but more intuitive quasi-notation of the λ-cube, which uses $[]^D$ for denoting holes of type or kind D.

i) The identity function over type A is a legal expression:

$$X:*\ \vdash_{\lambda[\to]}\quad (\lambda x{:}X.\,x)\ :\ (X{\to}X)\ :\ *.$$

ii) Consider the following examples:

$$\begin{array}{rl}
X:*,Y:*\vdash_{\lambda[\to]} & (\Lambda x{:}X,y{:}Y.\,x)\ :\ (\underline{\Lambda}x{:}X,y{:}Y.\,X)\ :\ \triangle_*\\
X:*,Y:*\vdash_{\lambda[P]} & (\Lambda x{:}X,y{:}Y.\,X)\ :\ (\underline{\Lambda}x{:}X,y{:}Y.*)\ :\ \triangle_\Box\\
\emptyset\vdash_{\lambda[2]} & (\Lambda X{:}*,x{:}X.\,x)\ :\ (\underline{\Lambda}X{:}*,x{:}X.\,X)\ :\ \triangle_*.
\end{array}$$

These examples involve elements that represent terms and types to be placed into a hole of a context. The element $\Lambda x{:}X,y{:}Y.\,x$ can be understood as the

term x which is to be placed into a hole in the scope of an x-abstractor and a y-abstractor. Hence the communication preamble $\Lambda x{:}X, y{:}Y$. Due to possible dependencies in the types as in the third example, the type of $\Lambda x{:}X, y{:}Y.\, x$ is the $\underline{\Lambda}$-abstraction $\underline{\Lambda} x{:}X, y{:}Y.\, X$. In comparison, in the applications of the context calculus, for example in $\lambda c^{\rightarrow}$, the type of the communicating term $\Lambda x{:}X, y{:}Y.\, x$ would be $[X, Y]X$.

Analogously, the other two statements can be explained. We mention only that we now deal with also communicating types (for example, $\Lambda x{:}X, y{:}Y.\, X$), and with types in communication (for example, X in $\Lambda X{:}*, x{:}X.\, x$). Such elements are not considered in the context calculus.

iii) Hole variables are of a communicating type or of a communicating kind; for example:

$$\emptyset \;\vdash_{\lambda[P\underline{\omega}]}\; h \;:\; (\underline{\Lambda} X{:}*\,.\,\Pi x{:}X.\,*) \;:\; \triangle_{\Box}.$$

iv) The example

$$\emptyset \;\vdash_{\lambda[P\underline{\omega}]}\; (\underline{\delta} h{:}(\underline{\Lambda} X'{:}*\,.\,*).\,\Pi X{:}*\,.\,\Pi x{:}\,h\,\langle X\rangle\,.\,*) \;:\; \Box$$

can be understood as a representation of the meta-context $\Pi X{:}*\,.\,\Pi x{:}[]^{*}.\,*$. The hole variable in $\Pi X{:}*\,.\,\Pi x{:}\,h\,\langle X\rangle\,.\,*$ is abstracted by a $\underline{\delta}$-abstraction because $\Pi X{:}*\,.\,\Pi x{:}[]^{*}.\,*$ is a kind with a hole.

v) The example

$$\emptyset \;\vdash_{\lambda[P2]}\; (\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,h\,\langle\rangle) \;:\; *$$

can be understood as a representation of the meta-context $[]^{*}$. Because the meta-context $[]^{*}$ is considered here as a type with a hole, its representation is a $\underline{\delta}$-abstraction. Note that the communication of the hole $[]^{*}$ is void: hence, the 'empty' multiple abstraction $\underline{\Lambda}\epsilon.\,*$ as the type of h and the 'empty' multiple application $h\,\langle\rangle$ as the representation of the hole.

vi) An example involving hole abstractions is

$$X:*\;\vdash_{\lambda[\rightarrow]}\; (\delta h{:}(\underline{\Lambda} y{:}X.\,X).\,\lambda x{:}X.\,h\,\langle x\rangle) \;:\; (\underline{\delta} h{:}(\underline{\Lambda} y{:}X.\,X).\,X{\rightarrow}X) \;:\; *.$$

This example can be understood as a representation of the typing statement $X:*\,\vdash\,(\lambda x{:}X.\,[]^{X}) : X{\rightarrow}X$. In comparison, in the applications of the context calculus, for example in $\lambda c^{\rightarrow}$, this typing statement would be represented as $X:*\,\vdash\,(\delta h{:}([X]X).\,\lambda x{:}X.\,h\,\langle x\rangle) \;:\; ([X]X \Rightarrow X{\rightarrow}X)$.

vii) The example

$$\emptyset \;\vdash_{\lambda[P2]}\; (\delta h{:}(\underline{\Lambda}\epsilon.\,*).\,\lambda x{:}\,h\,\langle\rangle\,.\,x) \;:\; (\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,\Pi x{:}\,h\,\langle\rangle\,.\,h\,\langle\rangle) \;:\; *$$

can be understood as a representation of the typing statement $\emptyset \vdash (\lambda x{:}[]^*.\, x) : (\Pi x{:}[]^*.\, []^*) : *$.

Note that the meta-context $\lambda x{:}[]^*.\, x$ (thus also, the expression $\delta h{:}(\underline{\Lambda}\epsilon.*).\, \lambda x{:}\, h\langle\rangle\, .\, x$) can be treated as a 'polymorphic' version of the identity function: by introducing a hole in a type, one implicitly states that 'for all types fitting the hole' $\lambda x{:}[]^*.\, x$ is of type $\Pi x{:}[]^*.\, []^*$.

viii) An example of a function over meta-contexts is the function that takes a 'polymorphic' identity function as an argument and instantiates it to the identity function over X:

$$\begin{array}{rl} X : * \vdash_{\lambda[P2]} & (\lambda c{:}(\underline{\delta} h{:}(\underline{\Lambda}\epsilon.*).\, \Pi x{:}\, h\langle\rangle\, .\, h\langle\rangle).\, c\lceil\Lambda\epsilon.\, X\rceil) \\ : & (\Pi c{:}(\underline{\delta} h{:}(\underline{\Lambda}\epsilon.*).\, \Pi x{:}\, h\langle\rangle\, .\, h\langle\rangle).\, \Pi x{:}X.\, X). \end{array}$$

ix) The meta-context of Example 6.1.1 can be represented as

$$Y : * \vdash_{\lambda[\omega]} (\delta h{:}(\underline{\Lambda}X{:}*\,.*).\, (\lambda X{:}*\,.\,\lambda x{:}\, h\langle X\rangle\, .\, x)Y)\lceil\Lambda X{:}*\,.\,X\rceil \;:\; (\Pi x{:}Y.\, Y).$$

Consider the fragment of its derivation where

$$\frac{\begin{array}{c} Y : *, h : (\underline{\Lambda}X{:}*\,.*) \vdash Y : * \\ Y : *, h : (\underline{\Lambda}X{:}*\,.*) \vdash (\lambda X{:}*\,.\,\lambda x{:}\, h\langle X\rangle\, .\, x) \;:\; (\Pi X{:}*\,.\,\Pi x{:}\, h\langle X\rangle\, .\, h\langle X\rangle) \end{array}}{Y : *, h : (\underline{\Lambda}X{:}*\,.*) \vdash (\lambda X{:}*\,.\,\lambda x{:}\, h\langle X\rangle\, .\, x)Y \;:\; (\Pi x{:}\, h\langle Y\rangle\, .\, h\langle Y\rangle).}$$

Note that, in contrast to the case in Example 6.1.1, the typing is preserved (modulo context-related rewriting) under the substitution $[\![h := \Lambda X{:}*\,.\,X]\!]$. The reduct of the element $(\lambda X{:}*\,.\,\lambda x{:}\, h\langle X\rangle\, .\, x)Y$ is of the same type

$$Y : *, h : (\underline{\Lambda}X{:}*\,.*) \vdash_{\lambda[\omega]} (\lambda x{:}\, h\langle Y\rangle\, .\, x) \;:\; (\Pi x{:}\, h\langle Y\rangle\, .\, h\langle Y\rangle).$$

Furthermore, one may check that all rewrite sequences starting from the expression $(\delta h{:}(\underline{\Lambda}X{:}*\,.*).\, (\lambda X{:}*\,.\,\lambda x{:}\, h\langle X\rangle\, .\, x)Y)\lceil\Lambda X{:}*\,.\,X\rceil$ end in $\lambda x{:}Y.\, x$.

x) By introducing a hole in the type of an another hole, the latter hole becomes 'polymorphic'. Consider the previous example with a hole in the type of h:

$$\begin{array}{l} X : *, a : X, \\ H : (\underline{\Lambda}x{:}X.*) \vdash_{\lambda[P]} \quad (\delta h{:}(\underline{\Lambda}x{:}X.\, H\langle x\rangle).\, h\langle a\rangle) \;:\; (\underline{\delta} h{:}(\underline{\Lambda}x{:}X.\, H\langle x\rangle).\, H\langle a\rangle). \end{array}$$

See also Section 6.5, where such 'polymorphic' holes are used for representing segments and mathematical structures.

Composition can be encoded into the systems of the context cube: the composition expression *comp* defined in the next definition handles two contexts with one hole each. The expression *comp* is rather involved; for a detailed explanation of the encoding, see Remark 3.1.9. Also alternative notions of composition are possible: in Section 6.5 we will treat composition (encoded as the expression *comp2*) of two segments with 'polymorphic' types.

Definition 6.3.10 (Composition) Let C and D be such that for a certain Γ

$$\begin{array}{lcll}\Gamma & \vdash & C & : & (\underline{\delta}h : (\underline{\Lambda}\vec{x}_1 : \vec{A}_1.\,\Pi\vec{x}_2 : \vec{A}_2.\,A).\,\Pi\vec{x}_3 : \vec{A}_3.\,E)\\ \Gamma & \vdash & D & : & (\underline{\Lambda}\vec{x}_1 : \vec{A}_1.\,\underline{\delta}k : (\underline{\Lambda}\vec{y}_1 : \vec{B}_1.\,\Pi\vec{y}_2 : \vec{B}_2.\,B).\,\Pi\vec{x}_2 : \vec{A}_2.\,A).\end{array}$$

Suppose $\vec{x}_1 \notin \text{FVAR}(\vec{B}_1, \vec{B}_2, B))$. Then $\underline{comp}\,CD$ is the composition of C and D where (the types of c, d, g are dropped for brevity, but they should be the same as in the type of $\underline{comp}$)

$$\begin{array}{lcll}\Gamma & \vdash & \underline{comp} & \equiv & \lambda c.\,\delta d.\,\delta g.\,c\lceil\Lambda\vec{x}_1 : \vec{A}_1.\,(d\langle\vec{x}_1\rangle)\lceil g\rceil\rceil\\ & : & & & \Pi c : (\underline{\delta}h : (\underline{\Lambda}\vec{x}_1 : \vec{A}_1.\,\Pi\vec{x}_2 : \vec{A}_2.\,A).\,\Pi\vec{x}_3 : \vec{A}_3.\,E).\\ & & & & \underline{\delta}d : (\underline{\Lambda}\vec{x}_1 : \vec{A}_1.\,\underline{\delta}k : (\underline{\Lambda}\vec{y}_1 : \vec{B}_1.\,\Pi\vec{y}_2 : \vec{B}_2.\,B).\,\Pi\vec{x}_2 : \vec{A}_2.\,A).\\ & & & & \underline{\delta}g : (\underline{\Lambda}\vec{y}_1 : \vec{B}_1.\,\Pi\vec{y}_2 : \vec{B}_2.\,B).\ \Pi\vec{x}_3 : \vec{A}_3.\,E\,.\end{array}$$

The expression $\underline{comp}$ is legal with respect to a particular system $\lambda[S]$ if C and D are legal expressions with respect to the same system $\lambda[S]$: a typing derivation of $\underline{comp}$ uses the same conditions and the same elements of $\mathcal{R}$ as the typing derivations of C and D.

Note that the untyped version of $\underline{comp}$, that is,

$$\lambda c.\,\delta d.\,\delta g.\,c\lceil\Lambda\vec{x}_1.\,(d\langle\vec{x}_1\rangle)\lceil g\rceil\rceil$$

is the same as composition in $\lambda c^{\rightarrow}$ (cf. $\underline{comp}$ in Section 4.2).

6.4 Properties of the context cube

In this section, we investigate the properties of the $\lambda[\,]$-cube. Technically speaking, the $\lambda[\,]$-cube is a simple extension of the λ-cube, which can easily be translated back into the λ-cube. Hence, the properties regarding typing and rewriting of the systems of the $\lambda[\,]$-cube can be stated and proved in the similar way as for the (corresponding) systems of the λ-cube.

The properties of (the systems of) the $\lambda[\,]$-cube can be categorised as follows:

- the properties of the $\lambda[\,]$-calculus,
- the relation between the $\lambda[\,]$-calculus and the underlying calculus which is defined on pseudo-expressions of the λ-cube;
- the properties of the typing system(s) of the $\lambda[\,]$-cube;
- the relation between the systems of the $\lambda[\,]$-cube and the systems of the λ-cube; and
- the properties of rewriting in the systems of $\lambda[\,]$-cube.

This section is divided into five subsections accordingly.

Remark 6.4.1 The proofs of the statements on typing are performed in the same way as in the λ-cube. Such a piece of technical luck is a consequence of the fact that the $\lambda[\,]$-cube can be seen as a macro over the λ-cube. In particular, the $\lambda[\,]$-cube does not introduce (β-)rewriting on a level where otherwise no β-rewriting is allowed in the λ-cube. This is comparable to the situation in [SP94], and contrary to the situation in for example [KBN99] and some Automath systems (see for example [NGdV94]), where due to the presence of applicative expressions on the level of types and kinds, other more powerful techniques in proofs are used.

Properties of the $\lambda[\,]$-calculus

Rewriting in the $\lambda[\,]$-calculus, that is, the rewriting on pseudo-expressions of the $\lambda[\,]$-cube, is confluent. This property will be used later in the proof that the rewriting on legal expressions in each of the eight systems of the $\lambda[\,]$-cube is confluent.

Theorem 6.4.2 (Confluence of the $\lambda[\,]$-calculus)
Rewriting in the $\lambda[\,]$-calculus is confluent.

Proof: The proof can be given by presenting the $\lambda[\,]$-calculus in a higher-order rewriting format, and then showing that the higher-order rewrite system related to the $\lambda[\,]$-calculus is orthogonal and hence confluent. This proof is analogous to the proof that the context calculus λc is confluent, which has been given in Section 3.2.5.
QED

The following lemma will be needed later in this section, in the proof of the Substitution lemma 6.4.11 for the systems of the $\lambda[\,]$-cube.

Lemma 6.4.3 (Substitution for pseudo-expressions)
Let A, $\vec{B}$, $\vec{C}$ be pseudo-expressions, let all $\vec{x}, \vec{y}$ be distinct and let $\vec{x} \notin \mathrm{FVar}(\vec{C})$. Then,

$$A\,[\![\vec{x} := \vec{B}]\!]\,[\![\vec{y} := \vec{C}]\!] = A\,[\![\vec{y} := \vec{C}]\!]\,[\![\vec{x} := \vec{B}[\![\vec{y} := \vec{C}]\!]]\!].$$

Proof: The proof is done by a straightforward induction to the structure of A.
QED

The $\lambda[\,]$-calculus and the underlying calculus of the λ-cube

We define a translation from the $\lambda[\,]$-calculus into the underlying calculus of the λ-cube. We also show that the translation preserves rewriting between pseudo-expressions.

Definition 6.4.4 (Translation to the calculus of the λ-cube)

i) Define $\llbracket\,\rrbracket$ as

$$\begin{array}{lcl}
\llbracket s \rrbracket & = & \begin{cases} s & \text{if } s \in \{*, \Box\} \\ s' & \text{if } s \equiv \triangle_{s'} \end{cases} \\
\llbracket x \rrbracket & = & x \\
\llbracket\, \lambda x : A.\, B \,\rrbracket & = & \lambda x : \llbracket A \rrbracket . \llbracket B \rrbracket \\
\llbracket\, A\, B \,\rrbracket & = & \llbracket A \rrbracket \llbracket B \rrbracket \\
\llbracket\, \Pi x : A.\, B \,\rrbracket & = & \Pi x : \llbracket A \rrbracket . \llbracket B \rrbracket \\
\llbracket\, \Lambda \vec{x} : \vec{A}.\, B \,\rrbracket & = & \lambda x_1 : \llbracket A_1 \rrbracket . \ldots \lambda x_n : \llbracket A_n \rrbracket . \llbracket B \rrbracket \\
\llbracket\, A \langle \vec{B} \rangle \,\rrbracket & = & \llbracket A \rrbracket \llbracket B_1 \rrbracket \ldots \llbracket B_n \rrbracket \\
\llbracket\, \underline{\Lambda} \vec{x} : \vec{A}.\, B \,\rrbracket & = & \Pi x_1 : \llbracket A_1 \rrbracket . \ldots \Pi x_n : \llbracket A_n \rrbracket . \llbracket B \rrbracket \\
\llbracket\, \delta h : A.\, B \,\rrbracket & = & \lambda h : \llbracket A \rrbracket . \llbracket B \rrbracket \\
\llbracket\, A \lceil B \rceil \,\rrbracket & = & \llbracket A \rrbracket \llbracket B \rrbracket \\
\llbracket\, \underline{\delta} h : A.\, B \,\rrbracket & = & \Pi h : \llbracket A \rrbracket . \llbracket B \rrbracket .
\end{array}$$

ii) Let Γ be a pseudo-basis. Then $\llbracket \Gamma \rrbracket = \{(u : \llbracket A \rrbracket) \mid (u : A) \in \Gamma\}$.

Lemma 6.4.5 *Let A and B be two pseudo-expressions of the $\lambda[\,]$-cube. If $A \to B$ in the $\lambda[\,]$-calculus, then $\llbracket A \rrbracket \twoheadrightarrow \llbracket B \rrbracket$ in the λ-cube.*

Proof: The proof is conducted by induction to the structure of A. More specifically, the following can be shown: (i) if $A \to_\beta B$ then $\llbracket A \rrbracket \to_\beta \llbracket B \rrbracket$; (ii) if $A \to_{full} B$ then $\llbracket A \rrbracket \to_\beta \llbracket B \rrbracket$; and (iii) if $A \to_{\textcircled{m}\beta} B$ then $\llbracket A \rrbracket \twoheadrightarrow_\beta \llbracket B \rrbracket$. In the third case, the translation of a $\textcircled{m}\beta$-step is empty if the redex which is contracted is of the form $(\Lambda\epsilon.\, b)\,\langle\rangle$. QED

Typing properties of the systems of the $\lambda[\,]$-cube

We show that each system of the $\lambda[\,]$-cube has the subject reduction property. This is proved in an analogous way as for the λ-cube.

Lemma 6.4.6 (Free variables lemma) *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma \vdash A : B$. Then the following holds:*

i) All $x_1, \ldots, x_n$ are distinct.

ii) $\textsc{FVar}(A) \cup \textsc{FVar}(B) \subseteq \Gamma$.

iii) $\textsc{FVar}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ *for* $1 \leq i \leq n$.

Lemma 6.4.7 (Start lemma) *Let Γ be a legal basis. Then the following holds:*

i) $\Gamma \vdash * : \Box$.

ii) If $(x : A) \in \Gamma$ *then* $\Gamma \vdash x : A$.

Proof: One proves first that for all Γ', B and C if $\Gamma' \vdash B : C$, then both (i) and (ii) hold for Γ'. These proofs are conducted by induction to the length of the derivation. Then both parts of the lemma follow from the fact that Γ is a legal basis, which means that there are B and C such that $\Gamma \vdash B : C$. QED

Lemma 6.4.8 (Thinning lemma) *Let Γ and Γ' be legal bases with $\Gamma' \subseteq \Gamma$. If $\Gamma' \vdash A : B$, then $\Gamma \vdash A : B$.*

Proof: The proof is conducted by induction to the length n of the derivation $\Gamma' \vdash A : B$.

If $n = 1$ then the derivation is the axiom $\emptyset \vdash * : \Box$. Then $\Gamma \vdash * : \Box$ by the Start lemma 6.4.7(i).

Suppose now $\Gamma' \vdash A : B$ is of length $n > 1$. The proof proceeds by distinguishing the cases according to the last typing rule applied in this derivation. Here, we treat only two cases.

(var) Then, for $s = *, \Box$ and $x \notin \Gamma''$,

$$\frac{\Gamma'' \vdash B : s}{\Gamma'', x : B \vdash x : B.}$$

Because $(x : B) \in \Gamma' = \Gamma'', x : B$ and $\Gamma' \subseteq \Gamma$, also $(x : B) \in \Gamma$. Then by the Start lemma 6.4.7(ii), and the fact that Γ is legal, we have $\Gamma \vdash x : B$.

$(mabs)$ Then,

$$\frac{\Gamma', \vec{x} : \vec{A} \vdash b : B \qquad \Gamma' \vdash (\underline{\Lambda}\vec{x} : \vec{A}. B) : \triangle_s}{\Gamma' \vdash (\Lambda\vec{x} : \vec{A}. b) : (\underline{\Lambda}\vec{x} : \vec{A}. B).}$$

If $|\vec{x}| = 0$ then the statement follows directly by the induction hypothesis (applied twice) and the rule $(mabs)$.

Suppose $|\vec{x}| > 0$. Without loss of generality we assume that $\vec{x} \notin \Gamma$. Note that for each $1 \leq i \leq |\vec{x}|$ the basis $\Gamma', x_1 : A_1, \ldots, x_i : A_i$ is legal.

Claim: For all $1 \leq j \leq |\vec{x}|$ the basis $\Gamma, x_1 : A_1, \ldots, x_j : A_j$ is legal.

Proof of the claim: For each x_j one can trace its introduction in the derivation $\Gamma', \vec{x} : \vec{A} \vdash b : B$. Each x_j is introduced by either the rule (var) or the rule $(weak)$. Both rules have as an assumption $\Gamma', x_1 : A_1, \ldots, x_{j-1} : A_{j-1} \vdash A_j : s_j$.

Then we prove by (bounded) induction to j that the basis $\Gamma', x_1 : A_1, \ldots, x_j : A_j$ is legal.

If $j = 1$ then, because $\Gamma' \vdash A_1 : s_1$ and using the induction hypothesis of the lemma, it holds that $\Gamma \vdash A_1 : s_1$. By the rule (var), it holds $\Gamma, x_1 : A_1 \vdash x_1 : A_1$, and hence, the basis $\Gamma, x_1 : A_1$ is legal.

Let $1 \leq j < |\vec{x}|$. Suppose $\Gamma, x_1 : A_1, \ldots, x_j : A_j$ is legal. Because $\Gamma', x_1 : A_1, \ldots, x_j : A_j \vdash A_{j+1} : s_{j+1}$ and by the induction hypothesis of the lemma, it holds $\Gamma, x_1 : A_1, \ldots, x_j : A_j \vdash A_{j+1} : s_{j+1}$. By the rule (var), it holds $\Gamma, x_1 : A_1, \ldots, x_j : A_j, x_{j+1} : A_{j+1} \vdash x_{j+1} : A_{j+1}$, and hence, the basis $\Gamma, x_1 : A_1, \ldots, x_j : A_j, x_{j+1} : A_{j+1}$ is legal.

Note that the proof of the claim uses the induction hypothesis of the lemma.

Back to the proof of the lemma. By the induction hypothesis for $\Gamma', \vec{x} : \vec{A}$ and $\Gamma, \vec{x} : \vec{A}$, and using the left-hand side assumption, it holds $\Gamma, \vec{x} : \vec{A} \vdash b : B$. By the induction hypothesis for Γ' and Γ and using the right-hand side assumption, it holds $\Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}.\, B) : \triangle_s$. Then by the rule $(mabs)$ it holds $\Gamma \vdash (\Lambda\vec{x} : \vec{A}.\, b) : (\underline{\Lambda}\vec{x} : \vec{A}.\, B)$. QED

In the proof of the Thinning lemma, the property has been used (and proved) for some cases that if Γ' and Γ are legal bases with $\Gamma' \subseteq \Gamma$, and if $\Gamma', \vec{x} : \vec{A}$ is legal, then $\Gamma, \vec{x} : \vec{A}$ is legal too. In fact, using the Thinning lemma, this property holds for all such Γ' and Γ.

Corollary 6.4.9 *Let Γ' and Γ be legal bases with $\Gamma' \subseteq \Gamma$. Let $\Gamma', \vec{x} : \vec{A}$ be also a legal basis. Then $\Gamma, \vec{x} : \vec{A}$ is legal too.*

Lemma 6.4.10 (Generation lemma)

i) *If $\Gamma \vdash s : C$ then $s \equiv *$ and $C \equiv \Box$.*

ii) *If $\Gamma \vdash x : C$ then $\exists s \in \mathcal{S} \exists B$ such that $C =_{\beta c} B$, $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$.*

iii) *If $\Gamma \vdash (\Pi x : A.\, B) : C$ then $\exists (s_1, s_2) \in \mathcal{R}$ such that $\Gamma \vdash A : s_1$, $\Gamma, x : A \vdash B : s_2$ and $C \equiv s_2$.*

iv) *If $\Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}.\, B) : C$ then $\exists (s_i, s) \in \mathcal{R}$ for $1 \leq i \leq |\vec{x}|$ such that $\Gamma, x_1 : A_1, \ldots, x_{i-1} : A_{i-1} \vdash A_i : s_i$, $\Gamma, \vec{x} : \vec{A} \vdash B : s$ and $C \equiv \triangle_s$.*

v) *If $\Gamma \vdash (\underline{\delta} h : A.\, B) : C$ then $\exists (s_1, s_2) \in \mathcal{R}$ such that $\Gamma \vdash A : \triangle_{s_1}$, $\Gamma, h : A \vdash B : s_2$ and $C \equiv s_2$.*

vi) *If $\Gamma \vdash (\lambda x : A.\, b) : C$ then $\exists s \in \{*, \Box\}\ \exists B$ such that $\Gamma \vdash (\Pi x : A.\, B) : s$, $\Gamma, x : A \vdash b : B$ and $C =_{\beta c} (\Pi x : A.\, B)$.*

vii) *If $\Gamma \vdash (F\, a) : C$ then $\exists A, B$ such that $\Gamma \vdash F : (\Pi x : A.\, B)$, $\Gamma \vdash a : A$ and $C =_{\beta c} B[\![x := a]\!]$.*

viii) *If $\Gamma \vdash (\Lambda\vec{x} : \vec{A}.\, b) : C$ then $\exists s \in \{*, \Box\}\ \exists B$ such that $\Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}.\, B) : \triangle_s$, $\Gamma, \vec{x} : \vec{A} \vdash b : B$ and $C =_{\beta c} (\underline{\Lambda}\vec{x} : \vec{A}.\, B)$.*

ix) *If $\Gamma \vdash (F\langle\vec{a}\rangle) : C$ then $\exists \vec{A}, B$ such that $\Gamma \vdash F : (\underline{\Lambda}\vec{x} : \vec{A}.\, B)$, $\Gamma \vdash a_i : A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!]$ and $C =_{\beta c} B[\![\vec{x} := \vec{a}]\!]$.*

x) *If* $\Gamma \vdash (\delta h : A.b) : C$ *then* $\exists s \in \{*, \Box\}\ \exists B$ *such that* $\Gamma \vdash (\underline{\delta} h : A.B) : s$, $\Gamma, h : A \vdash b : B$ *and* $C =_{\beta c} (\underline{\delta} h : A.B)$.

xi) *If* $\Gamma; \Sigma \vdash (F\lceil a \rceil) : C$ *then* $\exists A, B$ *such that* $\Gamma \vdash F : (\underline{\delta} h : A.B)$, $\Gamma \vdash a : A$ *and* $C =_{\beta c} B[\![h := a]\!]$.

Proof: Let $\Gamma \vdash A : C$ be a derivation corresponding with one of the cases of this lemma. In this derivation, there is the (lowest) step in the derivation where the expression A is formed, say $\Gamma' \vdash A : C'$. Then, between $\Gamma' \vdash A : B'$ and $\Gamma \vdash A : C$ only applications of the rules (*weak*) and (*conv*) may occur. These steps may only enlarge the basis Γ' into Γ, or perform a conversion in C', yielding C.

The lemma is then proved by considering the rule or axiom by which A is formed. The proof relies on the Thinning lemma and its Corollary 6.4.9.

We show only the case where $A \equiv \Lambda \vec{x} : \vec{A}.b$. Then A is formed by the step

$$\frac{\Gamma', \vec{x} : \vec{A} \vdash b : B \qquad \Gamma' \vdash (\underline{\Lambda} \vec{x} : \vec{A}.B) : \triangle_s}{\Gamma' \vdash (\Lambda \vec{x} : \vec{A}.b) : (\underline{\Lambda} \vec{x} : \vec{A}.B)}$$

where $C =_{\beta c} (\underline{\Lambda} \vec{x} : \vec{A}.B)$ and $\Gamma' \subseteq \Gamma$.

Using the Thinning lemma and the right-hand side assumption of the step, it holds $\Gamma \vdash (\underline{\Lambda} \vec{x} : \vec{A}.B) : \triangle_s$. With Corollary 6.4.9, the basis $\Gamma, \vec{x}$ is legal. Using the left-hand side assumption and the Thinning lemma, we have $\Gamma, \vec{x} : \vec{A} \vdash b : B$. In sum, we have $\Gamma \vdash (\underline{\Lambda} \vec{x} : \vec{A}.B) : \triangle_s$, $\Gamma, \vec{x} : \vec{A} \vdash b : B$ and $C =_{\beta c} (\underline{\Lambda} \vec{x} : \vec{A}.B)$ for certain B and $s \in \{*, \Box\}$. QED

Lemma 6.4.11 (Substitution lemma)

If $\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash b : B$ *and if* $\Gamma \vdash a_i : A_i[\![x_1 := a_1, \ldots, x_{i-1} := a_{i-1}]\!]$ *for* $1 \leq i \leq |\vec{a}|$, *then*

$$\Gamma, (\Gamma'[\![\vec{x} := \vec{a}]\!]) \vdash (b[\![\vec{x} := \vec{a}]\!]) : (B[\![\vec{x} := \vec{a}]\!]).$$

Proof: The statement is proved by induction to the length n of the derivation $\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash b : B$, and by distinguishing the cases according to the last rule applied in the derivation. The proof uses also the Substitution lemma for pseudo-expressions 6.4.3.

Here, we consider only two cases for the last rule of the derivation. By U^* we denote $U[\![\vec{x} := \vec{a}]\!]$.

(*mabs*) Then,

$$\frac{\Gamma, \vec{x} : \vec{A}, \Gamma', \vec{y} : \vec{C} \vdash b : B \qquad \Gamma, \vec{x} : \vec{A}, \Gamma' \vdash (\underline{\Lambda} \vec{y} : \vec{C}.B) : \triangle_s}{\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash (\Lambda \vec{y} : \vec{C}.b) : (\underline{\Lambda} \vec{y} : \vec{C}.B).}$$

By the induction hypothesis, using twice, and by the rule (*mabs*), we have

$$\frac{\Gamma, \Gamma'^* \vec{y} : \vec{C}^* \vdash b^* : B^* \qquad \Gamma, \Gamma'^* \vdash (\underline{\Lambda} \vec{y} : \vec{C}^*.B^*) : \triangle_s}{\Gamma, \Gamma'^* \vdash (\Lambda \vec{y} : \vec{C}^*.b^*) : (\underline{\Lambda} \vec{y} : \vec{C}^*.B^*),}$$

that is, by the definition of substitution

$$\Gamma, \Gamma'^* \vdash (\Lambda \vec{y} : \vec{C}.\, b)^* : (\underline{\Lambda} \vec{y} : \vec{C}.\, B)^*.$$

(*mapp*) Then, for $1 \leq i \leq |\vec{c}|$

$$\frac{\begin{array}{c}\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash F : (\underline{\Lambda} \vec{y} : \vec{C}.\, B) \\ \Gamma, \vec{x} : \vec{A}, \Gamma' \vdash c_i : C_i[\![y_1 := c_1, ..., y_{i-1} := c_{i-1}]\!]\end{array}}{\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash F\langle \vec{c} \rangle : B\,[\![\vec{y} := \vec{c}]\!]\,.}$$

By the induction hypothesis, used $|\vec{c}| + 1$ times, the rule (*mapp*) and the definition of substitution, we have

$$\frac{\Gamma, \Gamma'^* \vdash F^* : (\underline{\Lambda} \vec{y} : \vec{C}^*.\, B^*) \quad \Gamma, \Gamma'^* \vdash c_i^* : C_i^*[\![y_1 := c_1^*, \ldots, y_{i-1} := c_{i-1}^*]\!]}{\Gamma, \Gamma'^* \vdash (F\langle \vec{c} \rangle)^* : B^*\,[\![\vec{y} := \vec{c}^*]\!]\,.}$$

Finally, by the Substitution lemma for expressions, $B\,[\![\vec{x} := \vec{a}]\!]\,[\![\vec{y} := \vec{c}[\![\vec{x} := \vec{a}]\!]]\!]$ $= B\,[\![\vec{y} := \vec{c}]\!]\,[\![\vec{x} := \vec{a}]\!]$.

QED

Lemma 6.4.12 *Let $\Gamma \vdash A : B$. Then there is a sort s such that $B \equiv s$ or $\Gamma \vdash B : s$.*

Proof: The proof is done by induction to the length of derivation $\Gamma \vdash A : B$. The proof also uses the Generation lemma and the Substitution lemma. QED

Lemma 6.4.13 (Subject reduction)
If $\Gamma \vdash A : B$ and $A \to A'$ then $\Gamma \vdash A' : B$.

Proof: Let $\Gamma = \vec{x} : \vec{A}$ and $\Gamma' = \vec{x} : \vec{A}'$ with $|\vec{A}| = |\vec{A}'| = n$. We say that $\Gamma \to \Gamma'$ if there is $1 \leq i \leq n$ with $A_i \to A_i'$ and for all $1 \leq j \leq n$ with $j \neq i$, it holds $A_j = A_j'$.

One then proves the following two statements simultaneously by induction to the length of $\Gamma \vdash A : B$ and by distinguishing the last rule applied in this derivation:

i) If $\Gamma \vdash A : B$ and $A \to A'$ then $\Gamma \vdash A' : B$.

ii) If $\Gamma \vdash A : B$ and $\Gamma \to \Gamma'$ then $\Gamma' \vdash A : B$.

The proof relies on the Generation and Substitution lemmas, and on Lemma 6.4.12.

We show only the case where the last step in the derivation $\Gamma \vdash A : B$ is the application of the rule (*mapp*) and where A is the redex being contracted.

We have the following situation: for $1 \leq i \leq |\vec{a}| = |\vec{A}|$ (the assumptions are numbered for an easy reference)

$$\dfrac{\begin{array}{rl} \Gamma \vdash a_1 : A_1 & (1) \\ \vdots & \\ \Gamma \vdash a_i : A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!] & (i) \\ \vdots & \\ \Gamma \vdash a_n : A_n[\![x_1 := a_1, ..., x_{n-1} := a_{n-1}]\!] & (n) \\ \Gamma \vdash (\Lambda \vec{x} : \vec{C}.\, b) : (\underline{\Lambda} \vec{x} : \vec{A}.\, B) & (n+1) \end{array}}{\Gamma \vdash (\Lambda \vec{x} : \vec{C}.\, b)\langle \vec{a} \rangle : B[\![\vec{x} := \vec{a}]\!]}$$

and $(\Lambda \vec{x} : \vec{C}.\, b)\langle \vec{a} \rangle \to b[\![\vec{x} := \vec{a}]\!]$. With the assumption $(n+1)$ and the Generation lemma, there is D with $\Gamma \vdash (\underline{\Lambda} \vec{x} : \vec{C}.\, D) : \triangle_s$, $(\underline{\Lambda} \vec{x} : \vec{C}.\, D) =_{\beta c} (\underline{\Lambda} \vec{x} : \vec{A}.\, B)$ and $\Gamma, \vec{x} : \vec{C} \vdash b : D$. In the conversion, we have in particular $C_i =_{\beta c} A_i$ for all $1 \leq i \leq n$ and $D =_{\beta c} B$.

By the Generation lemma for $\Gamma \vdash (\underline{\Lambda} \vec{x} : \vec{C}.\, D) : \triangle_s$, there are $(s_i, s) \in \mathcal{R}$ for $1 \leq i \leq n$ such that $\Gamma, x_1 : C_1, \ldots, x_{i-1} : C_{i-1} \vdash C_i : s_i$ and $\Gamma, \vec{x} : \vec{C} \vdash D : s$.

> Claim: For all $1 \leq i \leq n$ it holds $\Gamma \vdash a_i : C_i[\![x_1 := a_1, \ldots, x_{i-1} := a_{i-1}]\!]$.
>
> Proof of the claim: We show this by bounded induction to i. Assume for all $j < i$ that $\Gamma \vdash a_j : C_j[\![x_1 := a_1, \ldots, x_{j-1} := a_{j-1}]\!]$. From $\Gamma, x_1 : C_1, \ldots, x_{i-1} : C_{i-1} \vdash C_i : s_i$ and by the induction hypothesis and the Substitution lemma we have $\Gamma, x_1 : C_1, \ldots, x_{i-1} : C_{i-1} \vdash C_i[\![x_1 := a_1, \ldots, x_{i-1} := a_{i-1}]\!] : s_i$. Consider the assumption (i), which says $\Gamma \vdash a_i : A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!]$. Because $C_i =_{\beta c} A_i$ and conversion is closed under substitution, also $C_i[\![x_1 := a_1, \ldots, x_{i-1} := a_{i-1}]\!] =_{\beta c} A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!]$. Using the rule $(conv)$ we arrive at $\Gamma \vdash a_i : C_i[\![x_1 := a_1, \ldots, x_{i-1} := a_{i-1}]\!]$, as required.

Because $\Gamma, \vec{x} : \vec{C} \vdash b : D$, and using the Substitution lemma with $[\![\vec{x} := \vec{a}]\!]$, it holds $\Gamma \vdash b[\![\vec{x} := \vec{a}]\!] : D[\![\vec{x} := \vec{a}]\!]$. Then with the rule $(conv)$,

$$\dfrac{\Gamma \vdash b[\![\vec{x} := \vec{a}]\!] : D\,[\![\vec{x} := \vec{a}]\!] \qquad \Gamma \vdash B[\![\vec{x} := \vec{a}]\!] : s \qquad B[\![\vec{x} := \vec{a}]\!] = D[\![\vec{x} := \vec{a}]\!]}{\Gamma \vdash b[\![\vec{x} := \vec{a}]\!] : B\,[\![\vec{x} := \vec{a}]\!]\,.}$$

QED

Typing in the $\lambda[\,]$-cube is not unique, because β-conversions is allowed in types. For example, if $\Gamma = \{X : *, x : X\}$ then both $\Gamma \vdash x : X$ and $\Gamma \vdash x : (\lambda Y : *.\, Y)\, X$. However, the uniqueness of typing does hold *up to conversion*.

The $\lambda[\,]$-cube and the λ-cube

The correspondence between the $\lambda[\,]$-calculus and the λ-cube can be summarised by two sentences:

- the system λS of the λ-cube can trivially be translated into the system $\lambda[S]$ of the $\lambda[\,]$-cube;
- the system $\lambda[S]$ of the $\lambda[\,]$-cube can be translated into the system λS of the λ-cube.

The translation of the system λS into the system $\lambda[S]$ is the identity function. The translation of the system $\lambda[S]$ into the system λS is given by Definition 6.4.4. Both translations preserve the notion of legal expression and of legal bases. Moreover, both translations preserve rewrite steps on legal expressions.

Lemma 6.4.14

i) If $\Gamma \vdash_{\lambda S} A : B$, *then* $\Gamma \vdash_{\lambda[S]} A : B$.

ii) If A *is a legal* λS*-expression and* $A \to_\beta A'$ *in the system* λS, *then* $A \to_\beta A'$ *in the system* $\lambda[S]$.

Proof: The proof of the first part follows trivially from the fact that the typing rules of the λ-cube are contained in the typing rules of the $\lambda[\,]$-cube, and from the fact that the system λS of the λ-cube and the system $\lambda[S]$ are parametrised by the same set of rules $\mathcal{R}$.

For the proof of the second part, note that, by the first part, A is also a legal $\lambda[S]$-expression. Then, the statement follows from the fact that the β-rewrite rule is also a rewrite rule of the system $\lambda[S]$ and from the closure property of rewriting in the system $\lambda[S]$ (i.e. the subject reduction property 6.4.13). QED

Lemma 6.4.15

i) If $\Gamma \vdash_{\lambda[S]} A : B$ *in the* $\lambda[\,]$*-cube, then* $⟦\Gamma⟧ \vdash_{\lambda S} ⟦A⟧ : ⟦B⟧$ *in the* λ*-cube.*

ii) If A *is a legal* $\lambda[S]$*-expression and* $A \to A'$, *then* $⟦A⟧ \twoheadrightarrow ⟦A'⟧$ *in the system* λS.

Proof: The proof of the first part is done by translating the typing rules: it can be shown that from the translations of the premisses, the translations of the conclusions can be derived in λS. The proof uses also Lemma 6.4.5 to show that if $B =_{\beta c} B'$ in the $\lambda[\,]$-cube, then $⟦B⟧ =_\beta ⟦B'⟧$ in the λ-cube.

The second part is a corollary of the first part of this lemma, Lemma 6.4.5 and the closure property of rewriting in the system λS (i.e. the subject reduction property 1.2.73 of λS). QED

Rewriting properties of the systems of the $\lambda[\,]$-cube

Rewriting in each of the systems of the $\lambda[\,]$-cube is confluent and strongly normalising. The confluence proof is a standard one; the proof of strong normalisation property is conducted via the translation into the λ-cube and the strong normalisation property of rewriting in the λ-cube.

Theorem 6.4.16 (Confluence of the $\lambda[\,]$-cube)
Rewriting in each system of the $\lambda[\,]$-cube is confluent.

Proof: The proof relies on the subject reduction property (Lemma 6.4.13) and the confluence property of the $\lambda[\,]$-calculus (Lemma 6.4.2). QED

Theorem 6.4.17 (Strong normalisation for the $\lambda[\,]$-cube)
Rewriting in any system of the $\lambda[\,]$-cube is strongly normalising.

Proof: Let A_0 be a legal expression of $\lambda[S]$ and suppose r is an infinite rewrite sequence in $\lambda[S]$:

$$r: \quad A_0 \to A_1 \to A_2 \to \ldots \ (\infty).$$

Note that due to the subject reduction property all reducts of A are legal expressions of $\lambda[S]$ too. Then, the translation (see Definition 6.4.4) of A_i's to the the system λS of the λ-cube results in a rewrite sequence $\llbracket r \rrbracket$ in λS:

$$\llbracket r \rrbracket: \quad \llbracket A_0 \rrbracket \twoheadrightarrow \llbracket A_1 \rrbracket \twoheadrightarrow \llbracket A_2 \rrbracket \twoheadrightarrow \ldots \ (\infty).$$

Because there are no infinite rewrite sequences in λS, the tail of $\llbracket r \rrbracket$ must eventually be empty, i.e. $\llbracket A_n \rrbracket \equiv \llbracket A_{n+1} \rrbracket \equiv \ldots$. These steps can only be translations of 'empty' $\textcircled{m}\beta$-steps, i.e. $A_n \equiv C[(\Lambda\epsilon.\, A)\,\langle\rangle] \to_{\textcircled{m}\beta} C[A] \equiv A_{n+1} \ldots$. Such tail of r is bounded by the size of A_n. Hence, there cannot be infinitely many such steps starting from A_n. Therefore, all reductions starting from A must be finite. QED

6.5 Application of the context cube: mathematical structures and segments

In this section we look more closely into the application of meta-contexts in proof checking and into the representation of this application within the $\lambda[\,]$-cube. The application of meta-contexts in proof checking has already been described as one of the motivations for formalisation of meta-contexts in Introduction and Section 2.3. It concerns the usage of meta-contexts for representing mathematical structures (see also Example 5.1.4). The meta-contexts employed are in fact 'polymorphic' (in the sense of Intermezzo 5.1.8) de Bruijn's segments with dependent types. When representing mathematical structures as segments, it is crucial that the types of holes are 'polymorphic', because only in this way a uniform representation can be obtained. With holes of 'polymorphic' types, such representations of mathematical structures can be used for different purposes, like for example in proofs of different lemmas involving the structure in question.

The application of meta-contexts in proof checking is illustrated by an example. From the mathematical point of view, the example treats a reflexive–euclidic relation and shows that such a relation is transitive and symmetric. From the technical point of view, the example shows how mathematical structures can be represented as meta-contexts, how such representations can be combined to form representations

of bigger mathematical structures, and how such representations can be used in proofs. First, we will work out the example involving meta-contexts in the informal, inadequate notation of the λ-cube. Then, we will 'translate' this example into the $\lambda[\,]$-cube. Finally, we will consider by way of an example representation of 'polymorphic' segments with dependent types in general within the $\lambda[\,]$-cube.

In this section we will sometimes, for the sake of brevity, write x^A in the abstractions instead of $x : A$.

Before going into details of the example, a remark on mathematical structures is in order. When dealing with mathematical structures, we distinguish between an abstract mathematical structure and a concrete mathematical structure. An abstract mathematical structure is a structure consisting of sets, relations, functions and assumptions about the properties of these, which may be considered as an entity in mathematics. For example, the structure of reflexive binary relation is $(S, R, \underline{ref} : (\Pi x : S.\, Rxx))$ where S is a set, R is a binary relation on S and $\underline{ref}$ is the assumption that R is reflexive. A concrete mathematical structure is a structure of concrete sets, functions and relations and proofs that they satisfy certain properties. For example, a concrete reflexive binary relation is $(\mathbb{N}, =, \underline{proof\_ref})$ where $\mathbb{N}$ is the set of natural numbers, the relation $=$ is the equality relation and $\underline{proof\_ref}$ is a proof (in natural language or as an expression of the lambda cube) that $=$ is reflexive.

In the example, we focus mainly on abstract mathematical structures.

An example with meta-contexts in informal notation

First we represent the structure of a binary relation as a segment and then extend it to a segment representing the structure of a reflexive–euclidic relation. Then we show how the segment representing the structure of a reflexive–euclidic relation can be used in the proof of the lemma which says that such a relation is also transitive.

In general, an abstract mathematical structure can be represented as a meta-context consisting of a sequence of abstractions which represent the components of the mathematical structure. In particular, the structure of a binary relation can be represented by the segment

$$\underline{\texttt{rel}} \;\equiv\; \lambda S : *.\, \lambda R : (S{\to}S{\to}*).\, [].$$

Furthermore, the properties of R being reflexive and euclidic can be represented as

$$\underline{\texttt{re}} \;\equiv\; \lambda \underline{ref} : (\Pi x : S.\, Rxx).\, \lambda \underline{euc} : (\Pi x, y, z : S.\, Rxy{\to}Rxz{\to}Ryz).\, [].$$

The segment $\underline{\texttt{re}}$ is designed in such a way that the intentional binding is expressed by the choice of the free variables S and R, which are intended to be bound by the binders of the segment $\underline{\texttt{rel}}$. Then the (meta-)composition of the two segments gives a representation of the structure of a reflexive–euclidic relation:

$$\begin{aligned}\underline{\texttt{re\_rel}} \;&\equiv\; \underline{\texttt{rel}}[\underline{\texttt{re}}] \equiv\\ &\lambda S : *.\, \lambda R : (S{\to}S{\to}*).\\ &\lambda \underline{ref} : (\Pi x : S.\, Rxx).\, \lambda \underline{euc} : (\Pi x, y, z : S.\, Rxy{\to}Rxz{\to}Ryz).\, [].\end{aligned}$$

We use these segments to prove that a reflexive–euclidic relation is transitive and symmetric. The proofs, which are built using a meta-operation of hole filling, and the lemma (i.e. the element and its type, respectively) are:

$$\begin{array}{rcl}\underline{\texttt{proof\_trans}} \equiv & \underline{\texttt{re\_rel}}\ [\lambda a,b,c:S.\,\lambda p:Rab.\,\lambda q:Rbc.\,\underline{euc}\,bac(\underline{euc}\,abap(\underline{ref}\,a))q] \\ : & \Pi S:*.\,\Pi R:(S{\to}S{\to}*). \\ & \Pi\underline{ref}:(\Pi x:S.\,Rxx).\,\Pi\underline{euc}:(\Pi x,y,z:S.\,Rxy{\to}Rxz{\to}Ryz). \\ & \Pi a,b,c:S.\,Rab\to Rbc\to Rac \\ \equiv & \underline{\texttt{trans\_lemma}}\end{array}$$

and

$$\begin{array}{rcl}\underline{\texttt{proof\_sym}} \equiv & \underline{\texttt{re\_rel}}\ [\lambda a,b,c:S.\,\lambda p:Rab.\,\underline{euc}\,abap(\underline{ref}\,a)] \\ : & \Pi S:*.\,\Pi R:(S{\to}S{\to}*). \\ & \Pi\underline{ref}:(\Pi x:S.\,Rxx).\,\Pi\underline{euc}:(\Pi x,y,z:S.\,Rxy{\to}Rxz{\to}Ryz). \\ & \Pi a,b:S.\,Rab\to Rba \\ \equiv & \underline{\texttt{sym\_lemma}}.\end{array}$$

Representing the example in the $\lambda[\,]$-cube

We represent the example given above into the $\lambda[\,]$-cube. All the expressions considered here are of course legal. Moreover, because all of the expressions are closed, they can be derived with an empty basis. For the sake of brevity, the type of the variables in abstractions will be left out if it is clear from the surrounding text what the types should be.

The segment $\underline{\texttt{rel}}$ representing the structure of a binary relation and the segment $\underline{\texttt{re}}$ representing the properties of the relation being reflexive and euclidic can be represented as the expressions $\underline{rel}$ and $\underline{re}$ in the $\lambda[\,]$-cube, where

$$\begin{array}{rcl}\underline{rel} \equiv & \delta H.\,\delta h.\,\lambda S.\,\lambda R.\,h\,\langle S,R\rangle \\ : & \underline{\delta}H^{\underline{\Lambda}S':*,R':(S'\to S'\to*).*}.\,\underline{\delta}h^{\underline{\Lambda}S':*,R':(S'\to S'\to*).H\,\langle S',R'\rangle}. \\ & \Pi S^{*}.\,\Pi R^{S\to S\to*}.\,H\,\langle S,R\rangle\end{array}$$

and

$$\begin{array}{rcl}\underline{re} \equiv & \Lambda S,R.\,\delta G.\,\delta g.\,\lambda\underline{ref}.\,\lambda\underline{euc}.\,g\,\langle S,R,\underline{ref},\underline{euc}\rangle \\ : & \underline{\Lambda}S^{*},R^{S\to S\to*}. \\ & \underline{\delta}G^{\underline{\Lambda}S':*,R':(S'\to S'\to*),r:(\Pi x:S'.R'xx),e:(\Pi x,y,z:S'.R'xy\to R'xz\to R'yz).*}. \\ & \underline{\delta}g^{\underline{\Lambda}S':*,R':(S'\to S'\to*),r:(\Pi x:S'.R'xx),e:(\Pi x,y,z:S'.R'xy\to R'xz\to R'yz).\,G\,\langle S',R',r,e\rangle}. \\ & \Pi\underline{ref}^{\Pi x:S.Rxx}.\,\Pi\underline{euc}^{\Pi x,y,z:S.Rxy\to Rxz\to Ryz}.\,G\,\langle S,R,\underline{ref},\underline{euc}\rangle\,.\end{array}$$

The novelty of this representation is the additional hole that captures the 'polymorphic' type of the hole of a segment. For example in $\underline{rel}$, the additional hole H enables us to say, informally, that the type of h is 'polymorphic', viz. $h : (\underline{\Lambda}S':*,R':(S{\to}S{\to}*).[])$. Note that this representation of segments is not a segment again, because it contains two holes.

Note also that, whenever the representation of a mathematical structure is used, first the type of this hole h has to be instantiated for the special purpose in question.

We proceed with the example. The composition of $\underline{rel}$ and $\underline{re}$ is represented by the expression $\underline{comp2}\ \underline{rel}\ \underline{re}$, where $\underline{comp2}$ is designed specifically for $\underline{rel}$ and $\underline{re}$:

$$\begin{array}{l}\underline{comp2} \equiv \\ \lambda c.\,\delta d.\,\delta K.\,\delta k.\,c\lceil \Lambda S, R.\,\Pi \underline{ref}.\,\Pi \underline{euc}.\,K\,\langle S, R, \underline{ref}, \underline{euc}\rangle\rceil\,\lceil \Lambda S, R.\,d\,\langle S, R\rangle\,\lceil K\rceil\,\lceil k\rceil\rceil\end{array}$$

where $K : (\underline{\Lambda} S', R', r, e.*)$ and $k : (\underline{\Lambda} S', R', r, e.\,K\,\langle S', R', r, e\rangle)$. Note that $\underline{comp2}$ defines a different kind of composition than $\underline{comp}$ of Definition 6.3.10. The difference is not only in the number of holes that the arguments of these compositions may have (one hole each in the case of $\underline{comp}$ versus two holes each in the case of $\underline{comp2}$), but also in the notion of composition that they capture. The notion of composition implemented by $\underline{comp}$ is a classical one, whereas the notion of composition captured by $\underline{comp2}$ is specifically designed for this kind of 'polymorphic' segments.

The composition $\underline{comp2}\ \underline{rel}\ \underline{re}$ reduces to $\underline{re\_rel}$ where

$$\begin{array}{ll} & \underline{re\_rel} \\ \equiv & \delta K.\,\delta k.\,\lambda S.\,\lambda R.\,\lambda \underline{ref}.\,\lambda \underline{euc}.\,k\,\langle S, R, \underline{ref}, \underline{euc}\rangle \\ : & \underline{\delta} K^{\underline{\Lambda} S':*, R':(S'\to S'\to *), r:(\Pi x:S'.\,R'xx), e:(\Pi x,y,z:S'.\,R'xy\to R'xz\to R'yz).*}. \\ & \underline{\delta} k^{\underline{\Lambda} S':*, R':(S'\to S'\to *), r:(\Pi x:S'.\,R'xx), e:(\Pi x,y,z:S'.\,R'xy\to R'xz\to R'yz).\,K\,\langle S', R', r, e\rangle}. \\ & \Pi S^*.\,\Pi R^{S\to S\to *}. \\ & \Pi \underline{ref}^{\Pi x:S.\,Rxx}.\,\Pi \underline{euc}^{\Pi x,y,z:S.\,Rxy\to Rxz\to Ryz}.\ K\,\langle S, R, \underline{ref}, \underline{euc}\rangle\ . \end{array}$$

The types $\underline{trans\_lemma}$ and $\underline{sym\_lemma}$, and their respective elements $\underline{proof\_trans}$ and $\underline{proof\_sym}$ represent the statement that a reflexive–euclidic relation is transitive and symmetric, and their proofs:

$$\begin{array}{ll} & \underline{proof\_trans} \\ \equiv & \underline{re\_rel}\quad \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\,\Pi a, b, c{:}S.\,Rab\to Rbc\to Rac\rceil \\ & \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\,\lambda a, b, c{:}S.\,\lambda p{:}Rab.\,\lambda q{:}Rbc.\,\underline{euc}\,bac(\underline{euc}\,abap(\underline{ref}\,a))q\rceil \\ \\ : & \Pi S^*.\,\Pi R^{S\to S\to *}.\,\Pi \underline{ref}^{\Pi x;S.\,Rxx}.\,\Pi \underline{euc}^{\Pi x,y,z:S.\,Rxy\to Rxz\to Ryz}. \\ & \Pi a, b, c{:}S.\,Rab\to Rbc\to Rac \\ \equiv & \underline{trans\_lemma} \end{array}$$

and

$$\begin{array}{ll} & \underline{proof\_sym} \\ \equiv & \underline{re\_rel}\quad \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\,\Pi a, b, c{:}S.Rab\to Rba\rceil \\ & \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\,\lambda a, b, c{:}S.\,\lambda p{:}Rab.\,\underline{euc}\,abap(\underline{ref}\,a)\rceil \\ \\ : & \Pi S^*.\,\Pi R^{S\to S\to *}.\,\Pi \underline{ref}^{\Pi x:S.\,Rxx}.\,\Pi \underline{euc}^{\Pi x,y,z:S.\,Rxy\to Rxz\to Ryz}. \\ & \Pi a, b, c{:}S.Rab\to Rba \\ \equiv & \underline{sym\_lemma}. \end{array}$$

Note that the proofs are built in a compositional way, by using uniform representations of the mathematical structures, their composition and the heart of the proof itself.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_1 \vdash h\langle X,P\rangle \;:\; (\Pi x^X.\, H\langle X,P,x\rangle)} \qquad \dfrac{\vdots}{\Gamma_1 \vdash (fX) \;:\; X}}{\Gamma_1 \vdash h\langle X,P\rangle(fX) \;:\; H\langle X,P,(fX)\rangle}(app) \qquad \dfrac{\vdots}{\Gamma_0 \vdash Y : *}}{\Gamma_0 \vdash (\lambda X^*.\,\lambda P^{X\to *}.\, h\langle X,P\rangle(fX))Y \;:\; (\Pi P^{Y\to *}.\, H\langle Y,P,(fY)\rangle)}(app)$$

$$\begin{array}{l}\text{with}\\ \Gamma_0 \;=\; \{Y:*, f:(\Pi Z^*.\,Z), H{:}(\underline{\Lambda} X^*, P^{X\to *}, x^X.\,*),\\ \qquad h{:}(\underline{\Lambda} X^*, P^{X\to *}.\,\Pi x^X.\, H\langle X,P,x\rangle)\}\\ \Gamma_1 \;=\; \Gamma_0, X:*, P:X\to *\end{array}$$

Figure 6.3: A fragment of a derivation

Segments with polymorphic types in the $\lambda[\,]$-cube

The examples above involved in fact telescopes, that is, segments without applications, which are of the form $\lambda\vec{x}:\vec{A}.\,[]$. Here we consider an example of a segment with applications.

Let $S \equiv (\lambda X{:}*\,.\,\lambda P{:}(X\to *).\,[]\,(fX))Y$ where $Y:*$ and $f:(\Pi Z:*.\,Z)$. This segment is in the $\lambda[\,]$-cube represented as

$$Y:*, f:(\Pi Z{:}*\,.\,Z) \vdash_{\lambda[C]}$$

$$\delta H^{(\underline{\Lambda} X:*, P:(X\to *), x:X.\,*}.\, \delta h^{(\underline{\Lambda} X:*, P:(X\to *).\,\Pi x:X.\, H\langle X,P,x\rangle)}.\; (\lambda X^*.\,\lambda P^{X\to *}.\, h\langle X,P\rangle(fX))Y$$

$$: \underline{\delta} H^{(\underline{\Lambda} X:*, P:(X\to *), x:X.\,*}.\, \underline{\delta} h^{(\underline{\Lambda} X:*, P:(X\to *).\,\Pi x:X.\, H\langle X,P,x\rangle)}.\; \Pi P^{Y\to *}.\, H\langle Y,P,(fY)\rangle\,.$$

Figure 6.3 is a fragment of its derivation, which shows how the type of the segments keeps track of the applications in the segment.

Chapter 7

Future work

We set out two ideas for future work.

The first idea concerns adding labels in communication in order to allow for a more flexible way of communication between a context on the one hand, and expressions and contexts to be put into its holes on the other hand. For example, by using labels the ordering of arguments in communication can be avoided.

The second idea concerns an internal definition of subtyping. As we have illustrated in Chapters 5 and 6, contexts can be used in automated reasoning for representing mathematical structures, like relations, monoids, groups etc. Then, the aim is to define subtyping so that for example, a group can be used whenever a monoid is expected, because each group is a monoid. We sketch a definition of subtyping that makes use of the communication labels.

We describe the two ideas in turn. Although the descriptions are rather detailed, they are a sketch and it is possible that some adaptations are necessary.

7.1 Communication labels

In our approach to the formalisation of contexts in the context calculus λc and the context cube $\lambda[\,]$, communication is represented by multiple abstraction and multiple application. In a communication redex $(\Lambda\vec{x}.\,M)\langle\vec{N}\rangle$ the variables $\vec{x}$ and arguments $\vec{N}$ are matched positionally, that is, the i^{th} variable x_i is matched with the i^{th} argument N_i, for $1 \leq i \leq n = |\vec{x}| = |\vec{N}|$, so

$$(\Lambda\vec{x}.\,M)\langle\vec{N}\rangle \to_{\textcircled{m}\beta} M\,[\![x_1 := N_1, \ldots, x_n := N_n]\!]\,.$$

This communication mechanism can be improved by adding labels and allowing the variables and the arguments to be accessed by a label.

In this section, we will show how communication labels can be introduced to the context cube; labels can be introduced to the context calculus in a similar way. We will first introduce labels, and then describe two ways of communicating through labels: full and partial communication. Full communication is just a notational

convenience, while partial communication is a technical extension of the context cube. Introduction of labels has primarily an effect on the syntax of expressions, on the communication rewrite rule and on the typing rules involving communication; in the account given below we focus on these issues. Furthermore, we will compare this way of formalisation of communication with a formalisation of communication where holes are annotated with substitutions.

In the text below we drop the types of the variables in abstractions if they are irrelevant.

Labels. The variables in multiple abstractions and the arguments between the brackets in a multiple application can be labelled, as in

$$\underline{\Lambda} x_1^{\mathtt{a}_1}, \ldots, x_m^{\mathtt{a}_m} . A, \qquad \Lambda x_1^{\mathtt{a}_1}, \ldots, x_m^{\mathtt{a}_m} . M \qquad \text{and} \qquad P \langle N_1^{\mathtt{b}_1}, \ldots, N_n^{\mathtt{b}_n} \rangle .$$

Of course, labels may not be renamed. Furthermore, each x_i among $\vec{x}$ in $\underline{\Lambda}\vec{x}^{\vec{\mathtt{a}}}.A$ or $\Lambda\vec{x}^{\vec{\mathtt{a}}}.M$ and each N_j among $\vec{N}$ in $P\langle\vec{N}^{\vec{\mathtt{b}}}\rangle$ should have one, unique label. Such labels can easily be implemented, as we argue here. Instead of labelling variables and expressions, labels could be attached to the multiple abstractors (viz. $\underline{\Lambda}^{\vec{\mathtt{a}}}$ and $\Lambda^{\vec{\mathtt{a}}}$) and the multiple applicator (viz. $\langle\ \rangle^{\vec{\mathtt{b}}}$). The uniqueness of $\mathtt{a}_i$ among $\vec{\mathtt{a}}$ and of $\mathtt{b}_j$ among $\vec{\mathtt{b}}$ can be ensured by the choice of the constructors. Then precisely one, unique unrenameable label is related to each variable among $\vec{x}$ and to each argument $\vec{N}$. With the labels being attached to the constructors, a valid labelling is preserved under rewriting.

Full communication. Full communication is generated by communication rewrite steps where all variables of the multiple abstraction Λ and all arguments between $\langle\ \rangle$ of the multiple application are used. More precisely, an expression of the form $(\Lambda\vec{x}^{\vec{\mathtt{a}}}.M)\langle\vec{N}^{\vec{\mathtt{b}}}\rangle$ is a communication redex provided that $\vec{\mathtt{b}}$ is a permutation of $\vec{\mathtt{a}}$, and

$$(\Lambda\vec{x}^{\vec{\mathtt{a}}}.M)\langle\vec{N}^{\vec{\mathtt{b}}}\rangle \to M\,[\![x_1 := N_{l_1}, \ldots, x_m := N_{l_m}]\!] \qquad (\textcircled{m}\beta)$$

where the label of x_i (i. e. $\mathtt{a}_i$) and the label of N_{l_i} (i. e. $\mathtt{b}_{l_i}$) are the same, for $1 \leq i \leq m$ and $1 \leq l_i \leq m$ with $m = |\vec{x}| = |\vec{N}|$. An example of a full communication rewrite step is

$$(\Lambda x_1^{\mathtt{a}_1}, x_2^{\mathtt{a}_2}, x_3^{\mathtt{a}_3} . M)\langle N_1^{\mathtt{a}_2}, N_2^{\mathtt{a}_1}, N_3^{\mathtt{a}_3}\rangle \to_{\textcircled{m}\beta} M\,[\![x_1 := N_2, x_2 := N_1, x_3 := N_3]\!] .$$

Partial communication. In the account above, an expression of the form $(\Lambda\vec{x}^{\vec{\mathtt{a}}}.M)\langle\vec{N}^{\vec{\mathtt{b}}}\rangle$ is a communication redex if $\vec{\mathtt{b}}$ is a permutation of $\vec{\mathtt{a}}$. In fact, this requirement can be loosened into the requirement that $\vec{\mathtt{a}}$ forms a selection of the labels of $\vec{\mathtt{b}}$, that is, that $\vec{\mathtt{a}}$ is a prefix of a permutation of $\vec{\mathtt{b}}$. Then, in effect, during a communication step the multiple abstraction selects among the arguments $\vec{N}$ via the labels:

$$(\Lambda\vec{x}^{\vec{\mathtt{a}}}.M)\langle\vec{N}^{\vec{\mathtt{b}}}\rangle \to M\,[\![x_1 := N_{l_1}, \ldots, x_m := N_{l_m}]\!] \qquad (\textcircled{m}\beta)$$

where the label of x_i (i. e. $\mathtt{a}_i$) and the label of N_{l_i} (i. e. $\mathtt{b}_{l_i}$) are the same, for $1 \leq i \leq m$ and $1 \leq l_i \leq n$ with $|\vec{x}| = m \leq n = |\vec{N}|$. An example of a partial communication rewrite step is

$$(\Lambda x_1^{\mathtt{a}_1}, x_2^{\mathtt{a}_2}.\, M)\langle N_1^{\mathtt{a}_2}, N_2^{\mathtt{a}_3}, N_3^{\mathtt{a}_1}, N_4^{\mathtt{a}_4}\rangle \to_{\textcircled{\scriptsize m}\beta} M\,[\![x_1 := N_3, x_2 := N_1]\!]\,.$$

The example below illustrates some complexity of partial communication.

Example 7.1.1 We give two examples of rewrite sequences.

i) Consider the rewrite sequence

$$\begin{array}{lll} s & \equiv & (\delta h.\,(h\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}\rangle)(h\langle P_1^{\mathtt{a}}, P_2^{\mathtt{b}}, P_3^{\mathtt{d}}\rangle))\lceil \Lambda x^{\mathtt{a}}.\, M\rceil \\ & \to_{\mathit{fill}} & ((\Lambda x^{\mathtt{a}}.\, M)\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}\rangle)((\Lambda x^{\mathtt{a}}.\, M)\langle P_1^{\mathtt{a}}, P_2^{\mathtt{b}}, P_3^{\mathtt{d}}\rangle) \\ & \to\!\!\!\!\to_{\textcircled{\scriptsize m}\beta} & (M[\![x := N_1]\!])(M[\![x := P_1]\!]). \end{array}$$

In s, the hole variable h occurs in two multiple applications with different labels. Consequently, the expression $\Lambda x^{\mathtt{a}}.\, M$, which is substituted for h, is involved in two different partial communication rewrite steps.

ii) Consider the rewrite sequence

$$\begin{array}{lll} t & \equiv & (\lambda y.\,(y\lceil \Lambda x_1^{\mathtt{a}}, x_2^{\mathtt{b}}.\, M\rceil)(y\lceil \Lambda x_1^{\mathtt{a}}, x_3^{\mathtt{c}}.\, P\rceil))(\delta h.\, h\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle) \\ & \to_{\beta} & ((\delta h.\, h\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle)\lceil \Lambda x_1^{\mathtt{a}}, x_2^{\mathtt{b}}.\, M\rceil) \\ & & ((\delta h.\, h\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle)\lceil \Lambda x_1^{\mathtt{a}}, x_2^{\mathtt{b}}.\, M\rceil) \\ & \to\!\!\!\!\to_{\mathit{fill}} & ((\Lambda x_1^{\mathtt{a}}, x_2^{\mathtt{b}}.\, M)\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle)((\Lambda x_1^{\mathtt{a}}, x_3^{\mathtt{c}}.\, P)\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle) \\ & \to\!\!\!\!\to_{\textcircled{\scriptsize m}\beta} & (M[\![x_1 := N_1, x_2 := N_2]\!])(P[\![x_1 := N_1, x_3 := N_3]\!]). \end{array}$$

In t, the variable y is applied to two multiple abstractions with different labels. This eventually leads to the involvement of the arguments $N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}$, which originate from the same expression $\delta h.\, h\langle N_1^{\mathtt{a}}, N_2^{\mathtt{b}}, N_3^{\mathtt{c}}, N_4^{\mathtt{d}}\rangle$, in two different partial communication rewrite steps.

Typing of the context cube with labels. In addition to the standard role of governing the functionality of expressions, typing in a system of the context cube with labelled communication should also control labels. Moreover, the typing rules should ensure that in expressions and their reducts only well-formed communication redexes occur. That is, the typing rules should also ensure that the requirement of full communication '$\vec{\mathtt{b}}$ is a permutation of $\vec{\mathtt{a}}$' and the requirement of the partial communication '$\vec{\mathtt{a}}$ is a prefix of a permutation of $\vec{\mathtt{b}}$' are fulfilled in all (existing and created) communication redexes along a rewrite sequence. Furthermore, the typing rules should guarantee that in a communication redex the variables $\vec{x}$ of the multiple abstraction and the arguments $\vec{N}$ of the multiple application agree on types: if a variable x_i among $\vec{x}$ and an argument N_j among $\vec{N}$ have the same label, they should be of the same type. Also, in the case of partial communication all arguments $\vec{N}$ should be legal.

Partial communication is stronger than full communication. Partial communication is stronger than full communication in the sense that full communication is a special case of partial communication, whereas the converse does not hold. As the former statement is straightforward, we focus the attention only on the latter. The only way to mimic partial communication by full communication is by adapting multiple abstractions or multiple applications so that only full communication redexes occur. There are two possibilities for adaptation: add fresh, dummy variables $\vec{y}$ in the multiple abstraction or discard superfluous arguments among $\vec{N}$. However, both attempts fail because permutation of arguments is allowed. This is witnessed by the examples of the rewrite sequences involving partial communication that were given in Example 7.1.1. In the expression s the multiple abstraction $\Lambda x^{\mathtt{a}}.\, M$ cannot be extended by dummies, and in the expression t the multiple application $h\langle N_1^{\mathtt{a}_1}, N_2^{\mathtt{a}_2}, N_3^{\mathtt{a}_3}\rangle$ cannot be adapted to fit both communication redexes that occur later in the respective reductions.

Flexible communication via labels. The effect of full and partial communication is the following.

In the case of formalisations of untyped or simply typed lambda calculus with contexts by adding labels, the order of arguments becomes irrelevant. One could consider the sequence of variables $\vec{x}$ as a set of variables $\{x_1, \ldots, x_m\}$ and the sequence of arguments $\vec{N}$ as a set of arguments $\{N_1, \ldots, N_n\}$. During the contraction of a communication redex, the variables and the arguments are matched through labels.

In the case of dependent typing, however, some ordering remains relevant, because the type of x_j may depend on x_i if $i < j$, for $1 \leq i, j \leq |\vec{x}|$. The orderings of variables $\vec{x}$ and arguments $\vec{N}$ which are allowed can be determined by studying the conditions under which the 'flattened' versions of such an expression, viz. $\Pi x_{i_1}{:}A_{i_1}. \ldots \Pi x_{i_n}{:}A_{i_n}.\, A$, $\lambda x_{i_1}{:}A_{i_1}. \ldots \lambda x_{i_n}{:}A_{i_n}.\, M$ and $P\, N_{j_1} \ldots N_{j_n}$ where $x_{i_1}, \ldots, x_{i_n}$ is a permutation of $\vec{x}$ and where $N_{j_1}, \ldots, N_{j_n}$ is a permutation of (in case of partial communication, a subset of) $\vec{N}$, are typable in a (corresponding) system of the context cube.

By adding labels into communication, one essentially annotates holes with substitutions. For example, the hole $[]^{[\![x:=M, y:=N]\!]}$ and the expression P to be put into the hole can be represented as $h\langle M^{\mathtt{x}}, N^{\mathtt{y}}\rangle$ and $\Lambda x^{\mathtt{x}}, y^{\mathtt{y}}.\, P$, respectively. Here $\mathtt{x}$ and $\mathtt{y}$ are (not renameable) labels, and x and y are (renameable) variables. On the representations rewriting is allowed, and, upon hole filling, the original substitution is reconstructed via the labels.

Other formalisations of contexts that employ holes annotated with substitutions are the calculus of M. Hashimoto and A. Ohori (see [HO98]), which has a labelled hole filling, and the calculus of I.A. Mason (see [Mas99]), which has holes labelled by substitutions. These context formalisations have already been discussed in Section 2.4 and Table 2.1. The context calculus of M. Hashimoto and A. Ohori allows only renamings, and in the formalism of I.A. Mason communication (as well as hole filling and composition) is formalised as a meta-operation.

7.2 Subtyping

As we have shown in Chapters 5 and 6, de Bruijn's segments can be used in automated reasoning for representing mathematical structures. Then, also statements about such structures can be formulated and proved. The flexibility and expressivity of automated reasoning about mathematical structures can be enhanced by supporting a notion of subtyping.

The notion of subtyping we have in mind is the one commonly used in mathematics. We explain this briefly by an example. Suppose $\underline{st\_is\_P}$ is a lemma which says that each symmetric–transitive relation Q has the property P:

$$\text{if } Qxy, Quv \text{ and } Qxu \text{ then } Qyv. \qquad (P)$$

Then the lemma $\underline{st\_is\_P}$ can be also applied to any relation R whose properties imply symmetry and transitivity. This is allowed because the same proof of the lemma can be used for such a relation R.

With mathematical structures represented as segments, the key question is how this treatment of mathematical structures and statements about them in mathematics translates into typing of segments in automated reasoning.

Remark 7.2.1 In the present setting of the context cube, where mathematical structures can be represented, the reasoning as above can be formalised, but only by explicitly proving the lemma anew for R. That is, in order to be able to apply the lemma $\underline{st\_is\_P}$ to a relation R whose properties imply symmetry and transitivity, one has to reproduce the proof of the lemma for the relation R.

Finding a typing system which also implements subtyping is here a key issue. In general, subtyping is induced from a relation between two segment types. We wish to say that S is a subtype of type T, denoted by $S \sqsubseteq T$, if whenever elements of T are used, elements of S can be used too[1].

Note that, the type of a segment in turn reflects the internal structure of the segment. Hence, this notion of subtyping is an internal one. As such, it is comparable to the definition of subtyping given by G. Betarte and A. Tasistro (see [BT95, BT97]). This notion of subtyping is in contrast to the (external) notion of subtyping as defined by J. Zwanenburg in [Zwa98]. There, the subtyping relation between types is postulated and it is not based on the internal structure of the objects of those types.

Subtyping by using communication labels

Communication labels can effectively be employed to implement a notion of subtyping. We present here an extended example of how subtyping via labels works. The example treats the lemma $\underline{st\_is\_P}$ about a symmetric–transitive relation that was mentioned above. The lemma will first be applied to a symmetric–transitive

[1] Such an informal definition of subtyping amounts to the standard one in object-oriented programming languages.

relation, and then to two relations whose properties include or entail symmetry and transitivity, namely to an equivalence relation and a reflexive–euclidic relation.

In the example we will concentrate on the communication steps and the types of the applications of the lemma to different relations. These types are an indicator how the subtyping relation should be defined.

The example is presented in the system $\lambda[C]$ of the context cube, which is understood as the system λC of the lambda cube with contexts (see Chapter 6 and in particular Section 6.5 for the representation technique employed). Technically speaking, the proof of the lemma is an expression and it is placed by hole filling into three different contexts. The communication steps that follow from hole filling involve partial communication. In the example, we will drop the types of many abstractions, for the sake of readability.

(1) Symmetric–transitive relation $\underline{st}$ and lemma $\underline{st\_is\_P}$

In this part of the example we present the representation of a symmetric–transitive relation and the representation of the lemma $\underline{st\_is\_P}$ and its proof.

The structure of a symmetric–transitive relation can be represented as

$$\begin{array}{lcl}
\underline{st} & \equiv & \delta H.\,\delta h.\,\lambda S.\,\lambda R.\,\lambda s.\,\lambda t.\,h\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle \\
& : & \underline{\delta} H : (\underline{\Lambda} S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}}.\,*). \\
& & \underline{\delta} h : (\underline{\Lambda} S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}}.\,H\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle). \\
& & \Pi S : *.\,\Pi R : (S{\to}S{\to}*). \\
& & \Pi s : (\Pi x, y{:}S.\,Rxy{\to}Ryx).\,\Pi t : (\Pi x, y, z{:}S.\,Rxy{\to}Ryz{\to}Rxz). \\
& & H\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle \\
& \equiv & T_{\underline{st}}.
\end{array}$$

In general, a statement and its proof involving a symmetric–transitive relation typically has the following form (the element corresponds to a proof, and the type to the statement):

$$(\lambda x.\,x\,\lceil\underline{\Lambda} A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\,T_M\rceil\,\lceil\Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\,M\rceil) \quad : \quad (\Pi x : T_{\underline{st}}.\,T).$$

Here, we assume that the type of $\Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\,M$ is $\underline{\Lambda} A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\,T_M$. Furthermore, the variable x stands for a symmetric–transitive relation and its type is the same as the type of $\underline{st}$, that is, its type is $T_{\underline{st}}$. The first argument of x 'instantiates' the type of the hole on the spine of x (the type of the hole h in $\underline{st}$) and the second argument is the expression that is actually placed in the hole on the spine of x. In the communication, the labels are the same as in $\underline{st}$, but we named the variables differently in order to emphasise how communication via labels works. We assume that each of the variables A, Q, $\underline{sym}$, and $\underline{tr}$ has the same type as the variable under the same label in $\underline{st}$. The abstractors Λ and $\underline{\Lambda}$, in a manner of speaking, unfold the binders $A^{\mathtt{S}}$, $Q^{\mathtt{R}}$, $\underline{sym}^{\mathtt{s}}$, $\underline{tr}^{\mathtt{t}}$ that are expected to occur, possibly renamed, in an expression substituted for x. In the expression M and its type T_M, the variables A, Q, $\underline{sym}$ and $\underline{tr}$ may freely be used.

We formalise the lemma $\underline{st\_is\_P}$ and its proof. Recall that the lemma says that if Q is a symmetric–transitive relation over the set A, then one can prove the following:

if Qxy and Quv and Qxu, then Qyv. The lemma and its proof are formalised as:

$$
\begin{array}{ll}
\underline{proof} & \\
\quad \equiv & \lambda x.\, x\, \lceil \underline{\Lambda} A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\, \Pi x,y,u,v{:}A.\, Qxy{\to}Quv{\to}Qxu{\to}Qyv \rceil \\
& \qquad \lceil \Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\, \lambda x,y,u,v{:}A.\, \lambda p_1{:}Qxy.\, \lambda p_2{:}Quv.\, \lambda p_3{:}Qxu. \\
& \qquad\quad \underline{tr}\, yuv(\underline{tr}\, yxu(\underline{sym}\, xyp_1)p_3)p_2 \rceil \\
\quad : & \Pi x{:}T_{\underline{st}}.\, \Pi S{:}*\,.\, \Pi R{:}(S{\to}S{\to}*).\, \Pi s{:}(\Pi x,y{:}S.\, Rxy{\to}Ryx). \\
& \Pi t{:}(\Pi x,y,z{:}S.\, Rxy{\to}Ryz{\to}Rxz).\, \Pi x,y,u,v{:}S.\;\; Rxy{\to}Ruv{\to}Rxu{\to}Ryv \\
\quad \equiv & \underline{st\_is\_P}.
\end{array}
$$

Obviously, this lemma can be applied to a symmetric–transitive relation $\underline{st}$. The application of the lemma amounts to the application of its proof $\underline{proof}$ to $\underline{st}$. This application, in turn, leads to the following reduction:

$$
\begin{array}{ll}
\underline{proof}\ \underline{st} & \\
\to_{\beta} & \delta H.\, \delta h.\, \lambda S.\, \lambda R.\, \lambda s.\, \lambda t.\, h\, \langle S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}} \rangle \\
& \lceil \underline{\Lambda} A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\, \Pi x,y,u,v{:}A.\, Qxy{\to}Quv{\to}Qxu{\to}Qyv \rceil \\
& \lceil \Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}.\, \lambda x,y,u,v{:}A.\, \lambda p_1{:}Qxy.\, \lambda p_2{:}Quv.\, \lambda p_3{:}Qxu. \\
& \quad \underline{tr}\, yuv(\underline{tr}\, yxu(\underline{sym}\, xyp_1)p_3)p_2 \rceil \\
& \\
\twoheadrightarrow_{\mathit{fill}} & \lambda S.\, \lambda R.\, \lambda s.\, \lambda t. \\
& (\Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}. \\
& \lambda x,y,u,v{:}A.\, \lambda p_1{:}Qxy.\, \lambda p_2{:}Quv.\, \lambda p_3{:}Qxu.\, \underline{tr}\, yuv(\underline{tr}\, yxu(\underline{sym}\, xyp_1)p_3)p_2) \\
& \langle S^{\mathtt{S}}, R^{\mathtt{R}}, s^{\mathtt{s}}, t^{\mathtt{t}} \rangle \\
& \\
\to_{\textcircled{\scriptsize m}\beta} & \lambda S.\, \lambda R.\, \lambda s.\, \lambda t. \\
& (\lambda x,y,u,v{:}A.\, \lambda p_1{:}Qxy.\, \lambda p_2{:}Quv.\, \lambda p_3{:}Qxu.\, \underline{tr}\, yuv(\underline{tr}\, yxu(\underline{sym}\, xyp_1)p_3)p_2)) \\
& \quad [\![A := S, Q := R, \underline{sym} := s, \underline{tr} := t]\!] \\
& \\
= & \lambda S.\, \lambda R.\, \lambda s.\, \lambda t. \\
& \lambda x,y,u,v{:}S.\, \lambda p_1{:}Rxy.\, \lambda p_2{:}Ruv.\, \lambda p_3{:}Rxu.\, t\, yuv(t\, yxu(s\, xyp_1)p_3)p_2.
\end{array}
$$

The communication step in this reduction involves full communication, because the labels $\mathtt{S}$, $\mathtt{R}$, $\mathtt{s}$, $\mathtt{t}$ in the multiple abstraction are the same as the labels in the multiple application. Note that, the final expression is a $\lambda[C]$-expression of type

$$\Pi S{:}*.\, \Pi R{:}(S{\to}S{\to}*).\, \Pi s{:}(\Pi x,y{:}S.\, Rxy{\to}Ryx).\, \Pi t{:}(\Pi x,y,z{:}S.\, Rxy{\to}Ryz{\to}Rxz).\; \Pi x,y,u,v{:}A.\;\; Rxy{\to}Ruv{\to}Rxu{\to}Ryv.$$

Hence, the expression $\underline{proof}\ \underline{st}$ should be of the same type.

(2) Applying $\underline{st\_is\_P}$ to equivalence relation $\underline{eq}$

We apply the lemma to an equivalence relation, which is reflexive, symmetric and transitive. The additional property (reflexivity) is not used in the proof of the

lemma. The structure of an equivalence relation is represented as

$$\begin{array}{lcl}
\underline{eq} & \equiv & \delta H.\,\delta h.\,\lambda S.\,\lambda R.\,\lambda r.\,\lambda s.\,\lambda t.\,h\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle \\
& : & \underline{\delta} H{:}(\underline{\Lambda} S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}.\,*). \\
& & \underline{\delta} h{:}(\underline{\Lambda} S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}.\,H\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle). \\
& & \Pi S{:}*\,.\,\Pi R{:}(S{\to}S{\to}*).\,\Pi r{:}(\Pi x{:}S.\,Rxx). \\
& & \Pi s{:}(\Pi x, y{:}S.\,Rxy{\to}Ryx).\,\Pi t{:}(\Pi x, y, z{:}S.\,Rxy{\to}Ryz{\to}Rxz). \\
& & H\,\langle S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle \\
& \equiv & T_{\underline{eq}}.
\end{array}$$

When designing a context calculus with subtyping, we would expect that $T_{\underline{eq}}$ is a subtype of $T_{\underline{st}}$, that is, $T_{\underline{eq}} \sqsubseteq T_{\underline{st}}$.

The application of (the proof of) the lemma above to $\underline{eq}$ leads to the following reduction:

$$\begin{array}{ll}
\underline{proof}\ \underline{eq} & \\
\to_\beta \twoheadrightarrow_{fill} & \lambda S.\,\lambda R.\,\lambda r.\,\lambda s.\,\lambda t. \\
& (\Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}. \\
& \lambda x, y, u, v{:}A.\,\lambda p_1{:}Qxy.\,\lambda p_2{:}Quv.\,\lambda p_3{:}Qxu.\,\underline{tr}\,yuv(\underline{tr}\,yxu(\underline{sym}\,xyp_1)p_3)p_2) \\
& \langle S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, s^{\mathtt{s}}, t^{\mathtt{t}}\rangle \\
& \\
\to_{\textcircled{m}\beta} & \lambda S.\,\lambda R.\,\lambda r.\,\lambda s.\,\lambda t. \\
& (\lambda x, y, u, v{:}A.\,\lambda p_1{:}Qxy.\,\lambda p_2{:}Quv.\,\lambda p_3{:}Qxu.\,\underline{tr}\,yuv(\underline{tr}\,yxu(\underline{sym}\,xyp_1)p_3)p_2) \\
& [\![A := S, Q := R, \underline{sym} := s, \underline{tr} := t]\!] \\
& \\
= & \lambda S.\,\lambda R.\,\lambda r.\,\lambda s.\,\lambda t. \\
& \lambda x, y, u, v{:}S.\,\lambda p_1{:}Rxy.\,\lambda p_2{:}Ruv.\,\lambda p_3{:}Rxu.\,t\,yuv(t\,yxu(s\,xyp_1)p_3)p_2.
\end{array}$$

The communication step involves partial communication. The multiple abstraction selects only the arguments labelled by `S`, `R`, `s`, and `t`, and drops the argument labelled by `r`. The final expression is a $\lambda[C]$-expression of type

$$\begin{array}{l}
\Pi S{:}*\,.\,\Pi R{:}(S{\to}S{\to}*).\,\Pi r{:}(\Pi x{:}S.\,Rxx). \\
\Pi s{:}(\Pi x, y{:}S.\,Rxy{\to}Ryx).\,\Pi t{:}(\Pi x, y, z{:}S.\,Rxy{\to}Ryz{\to}Rxz). \\
\Pi x, y, u, v{:}S.\ Rxy{\to}Ruv{\to}Rxu{\to}Ryv.
\end{array}$$

Hence, the type of $\underline{proof}\ \underline{eq}$ should be the same. Compared to the type of $\underline{proof}\ \underline{st}$, this type has an additional $\Pi r : (\Pi x{:}S.\,Rxx)$, as expected.

(3) Applying $\underline{st\_is\_P}$ to reflexive–euclidic relation $\underline{re\_st}$

The following example involves a relation with properties that imply symmetry and transitivity. This relation is a reflexive–euclidic binary relation and it was also treated in Example 5.1.4. In that example, we have shown that such a relation is also transitive and symmetric, by the tree depicted in the Figure 5.4. That tree can be reduced and represented in the present notation as:

$$\begin{array}{lcl}
\underline{re_st} & \equiv & \delta H.\,\delta h.\,\lambda S.\,\lambda R.\,\lambda r.\,\lambda e. \\
& & \quad h\,\langle\ S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, e^{\mathtt{e}}, \\
& & \quad\ (\lambda x, y, z{:}S.\,\lambda p{:}Rxy.\,\lambda q{:}Ryz.\,e\,yxz(e\,xyxp(r\,x))q)^{\mathtt{t}}, \\
& & \quad\ (\lambda x, y{:}S.\,\lambda p{:}Rxy.\,e\,xyxp(r\,x))^{\mathtt{s}}\ \rangle.
\end{array}$$

Note that $\underline{re\_st}$ has also labelled expressions in the multiple application, as opposed to only labelled variables as the previous examples had. Note also that this is a representation of a reflexive–euclidic relation with the communication extended by $\mathtt{s}$ and $\mathtt{t}$.

When designing a context calculus with subtyping, we would expect that the type $T_{\underline{re\_st}}$ of $\underline{re\_st}$ is a subtype of $T_{\underline{st}}$, that is, $T_{\underline{re\_st}} \sqsubseteq T_{\underline{st}}$.

The (proof of) lemma $\underline{st\_is\_P}$ can be applied to $\underline{re\_st}$, allowing the following reduction. We highlight only the partial communication step, and the final reduct:

$$
\begin{array}{ll}
\underline{proof}\ \underline{re\_st} & \\
\twoheadrightarrow_\beta\twoheadrightarrow_{fill} & \lambda S.\,\lambda R.\,\lambda r.\,\lambda e. \\
& (\Lambda A^{\mathtt{S}}, Q^{\mathtt{R}}, \underline{sym}^{\mathtt{s}}, \underline{tr}^{\mathtt{t}}. \\
& \lambda x,y,u,v{:}A.\,\lambda p_1{:}Qxy.\,\lambda p_2{:}Quv.\,\lambda p_3{:}Qxu.\,\underline{tr}\,yuv(\underline{tr}\,yxu(\underline{sym}\,xyp_1)p_3)p_2) \\
& \langle\, S^{\mathtt{S}}, R^{\mathtt{R}}, r^{\mathtt{r}}, e^{\mathtt{e}}, \\
& (\lambda x,y,z{:}S.\,\lambda p{:}Rxy.\,\lambda q{:}Ryz.\,e\,yxz(e\,xyxp(r\,x))q)^{\mathtt{t}}, \\
& (\lambda x,y{:}S.\,\lambda p{:}Rxy.\,e\,xyxp(r\,x))^{\mathtt{s}}\,\rangle \\
\\
\rightarrow_{\textcircled{m}\beta} & \lambda S.\,\lambda R.\,\lambda r.\,\lambda e. \\
& (\lambda x,y,u,v{:}A.\,\lambda p_1{:}Qxy.\,\lambda p_2{:}Quv.\,\lambda p_3{:}Qxu.\,\underline{tr}\,yuv(\underline{tr}\,yxu(\underline{sym}\,xyp_1)p_3)p_2) \\
& [\![A := S, Q := R, \\
& \underline{sym} := \lambda x,y{:}S.\,\lambda p{:}Rxy.\,e\,xyxp(r\,x), \\
& \underline{tr} := \lambda x,y,z{:}S.\,\lambda p{:}Rxy.\,\lambda q{:}Ryz.\,e\,yxz(e\,xyxp(r\,x))q]\!] \\
\\
\twoheadrightarrow_\beta & \lambda S.\,\lambda R.\,\lambda r.\,\lambda e. \\
& \lambda x,y,u,v{:}S.\,\lambda p_1{:}Rxy.\,\lambda p_2{:}Ruv.\,\lambda p_3{:}Rxu. \\
& e\,uyv(e\,yuy(e\,xyu(e\,yxy(e\,xyxp_1(r\,x))(r\,y))p_3)(r\,y))p_2.
\end{array}
$$

With a little bit of patience, one can see that the final expression is a $\lambda[C]$-expression of the type

$$
\begin{array}{l}
\Pi S{:}*\,.\,\Pi R{:}(S{\to}S{\to}*).\,\Pi r{:}(\Pi x{:}S.\,Rxx).\,\Pi e{:}(\Pi x,y,z{:}S.\,Rxy{\to}Rxz{\to}Ryz). \\
\Pi x,y,u,v{:}S.\,Rxy{\to}Ruv{\to}Rxu{\to}Ryv.
\end{array}
$$

Hence, the application $\underline{proof}\ \underline{re\_st}$ should be of the same type.

More on (sub)typing and partial communication

Some thoughts on subtyping and typing of partial communication, and their mutual dependence are presented.

The formalisation of a notion of subtyping as described by the example above is based on two essential ingredients:

- Polymorphism of segments: As discussed in Intermezzo 5.1.8, segment types are polymorphic because the types of holes are polymorphic: the type of a hole only states the minimal functionality that the hole has, but this type has an 'open end.' Consequently, the hole of a segment may be filled by any expression that agrees on its minimal functionality. For example, provided

they agree on the minimal functionality, a hole may be filled by proofs of different lemmas, or by segments to form representations of other, bigger mathematical structures. That means that one does not have to design a different segment, i.e. a different representation of a mathematical structure for each application. Hence, polymorphism of segments supports a uniform representation of mathematical structures as segments.

- Partial communication: Partial communication allows a flexible communication between an outer segment C on the one hand and an expression M or a segment D to be put into the hole (on the spine) of the outer segment on the other hand by allowing the expression M or the segment D to use only a necessary part of the communication offered by the outer segment C. The example above illustrated that a proof 'given in the context of' an equivalence relation can be 'placed into the context of' a relation that entails the properties of an equivalence relation. Hence, partial communication supports reusability of proofs.

Technically speaking, these two features are complementary in the following sense. Provided the requirements regarding minimal functionality and communication are satisfied, polymorphism allows different expressions to be placed into the same segment, whereas partial communication allows the same expression to be placed into different segments. On the level of applications in automated reasoning, polymorphism allows representation of mathematical structures in a uniform way and partial communication allows reusability of proofs.

With partial communication being one of the features for implementing subtyping, a design of a context calculus with subtyping tackles also the problems of typing partial communication.

The subtyping relation is induced from a relation between the types of two segments. A segment type carries also information about the communication labels of the hole of its elements. The subtyping relation between two segment types is determined by considering these communication labels and, in addition, by considering the types of the holes.

Once the subtyping relation between segment types is established, the relation can be extended to types of expressions, segments and functions involving segments. We expect this extension will resemble the definition of subtyping on records given by G. Betarte and A. Tasistro (see [BT95, BT97]).

Bibliography

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[Bal86] H. Balsters. *Lambda calculus extended with segments*. PhD thesis, Eindhoven University of Technology, 1986. Also in [NGdV94].

[Bal87] H. Balsters. Lambda calculus extended with segments. In *Mathematical logic and theoretical computer science (College Park, Md., 1984–1985)*, pages 15–27. Dekker, New York, 1987.

[Bar84] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, revised edition, 1984. (Second printing 1985).

[Bar92] H.P. Barendregt. Lambda calculi with types. In S. Abramski, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[Ben77] L.S. van Benthem Jutting. *Checking Landau's Grunlagen in the Automath system*. PhD thesis, Eindhoven University of Technology, 1977.

[Ben81] L.S. van Benthem Jutting. Description of AUT-68. Memorandum 1981-12, Dept. of Math., 1981.

[Ber78] G. Berry. Séquentialité de l'évaluation formelle des lambda-expressions. In B. Robinet, editor, *Transformations de Programmes; 3^e Colloque International sur la Programmation, Paris, France*, pages 66–80, Paris, 1978. Dunod.

[Ber88] S. Berardi. Towards a Mathematical Analysis of the Coquand–Huet Calculus of Constructions and Other Systems in Barendregt's Cube. Technical report, Dept. Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Università di Torino, 1988.

[Blo99] R. Bloo. Pure type systems with explicit substitution. Preprint available at `http://www.win.tue.nl/~bloo`, 1999.

[Bog95] M. Bognar. A survey of abstract rewriting. Master's Thesis, Vrije Universiteit Amsterdam, 1995.

[Bru67] N.G. de Bruijn. Verification of mathematical proofs by a computer, A preparatory study for a project Automath. In [NGdV94], 1967.

[Bru70] N.G. de Bruijn. The mathematical language Automath, its usage and some of its extensions. In M. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration, IRIA, Versailles,1968*, pages 29–61. Springer Verlag, Berlin, 1970. Also in Lecture Notes in Math. 125 and in [NGdV94].

[Bru71] N.G. de Bruijn. AUT-SL, a single line version of Automath. Notitie 71-72, Dept. of Math., 1971.

[Bru72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Math.*, 34:381–392, 1972.

[Bru78] N.G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report 78-WSK-03, Technological University Eindhoven, 1978.

[Bru80] N.G. de Bruijn. A survey of the project Automath. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York/London, 1980. Also in [NGdV94].

[Bru91] N.G. de Bruijn. Telescopic Mappings in Typed Lambda Calculus. *Information and Computation*, 91:189–204, 1991.

[BT95] G. Betarte and A. Tasistro. Formalisation of Systems of Algebras using Dependent Record Types and Subtyping: An Example. In *Proceedings of the 7th Nordic workshop on Programming Theory*, Göteborg, Sweden, 1995.

[BT97] G. Betarte and A. Tasistro. Extension of Martin–Löf's type theory with record types and subtyping. In *25 Years of Constructive Type Theory*. Oxford University Press, 1997.

[BV99] M. Bognar and R.C. de Vrijer. Segments in the context of contexts. Preprint available at `http://www.cs.vu.nl/~mirna`, 1999.

[BV01] M. Bognar and R.C. de Vrijer. A calculus of lambda calculus contexts. *Journal of Automated Reasoning*, 27(1):29–59, 2001.

[BV02] M. Bognar and R.C. de Vrijer. The context cube $\lambda[]$: Barendregt's lambda cube with contexts. Informal Proceedings of the Workshop on Thirty Five years of Automath, Edinburgh, 2002.

[CF58] H.B Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, Amsterdam, 1958.

[Coq] The Coq Proof Assistant. `http://pauillac.inria.fr/coq/`.

[CPM90] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Daa73] D.T. van Daalen. A description of Automath and some aspects of its language theory. In *Proceedings of the Symposium APLASM*, volume 1, pages 48–77, Orsay, 1973. Also in [NGdV94].

[Daa80] D.T. van Daalen. *The language theory of Automath*. PhD thesis, Eindhoven University of Technology, 1980.

[DPS97] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Typed lambda calculi and applications (Nancy, 1997)*, pages 147–163. Springer, Berlin, 1997.

[Geu93] H. Geuvers. *Logics and Type Systems*. PhD thesis, Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands, September 1993.

[GJ02] J.H. Geuvers and G.I. Jojgov. Open Proofs and Open Terms: a Basis for Interactive Logic. *Proceedings of CSL'02, LNCS*, 2471:537–552, 2002.

[GN91] J.H. Geuvers and M.J. Nederhof. A modular proof of strong normalisation for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[GS91] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[HL78] G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11:31–55, 1978.

[HO98] M. Hashimoto and A. Ohori. A typed context calculus. *Sūrikaisekikenkyūsho Kōkyūroku*, 1023:76–91, 1998. Type theory and its application to computer systems (Japanese) (Kyoto, 1997).

[HOL] The HOL Theorem Prover. `http://archive.comlab.ox.ac.uk/formal-methods/hol.html`.

[HS86] J.R. Hindley and J.P. Seldin. *An Introduction to Combinators and λ-Calculus*. London Mathemitical Society Student Texts. University Press, Cambridge, 1986.

[Isa] Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[Kah93] S. Kahrs. Context rewriting. In *Conditional term rewriting systems (Pont-à-Mousson, 1992)*, pages 21–35. Springer, Berlin, 1993.

[Kam81] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk, Th. Janssen, and M. Stokhof, editors, *Formal methods in the study of language*, pages 277–322. Mathematisch Centrum, Amsterdam, 1981.

[KBN99] F. Kamareddine, R. Bloo, and R. Nederpelt. On Pi-conversion in the lambda-cube and the combination with abbreviations. *Annals of Pure and Applied Logic*, 97:27–45, 1999.

[Kha90] Z.O. Khasidashvili. Expression reduction systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, 1990.

[KKM99] M. Kohlhase, S. Kuschert, and M. Müller. Dynamic lambda calculus. Preprint, available at `http://www.ags.uni-sb.de/~kohlhase`, 1999.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts Nr. 127. CWI, Amsterdam, 1980. PhD Thesis.

[Klo92] J.W. Klop. Term Rewriting Systems. In D. Gabbay S. Abramski and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[KOR93] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory Reduction Systems, Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993.

[LEG] The LEGO Proof Assistant. `http://www.dcs.ed.ac.uk/home/lego/`.

[LF96] S.-D. Lee and D. Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1996.

[Mag96] L. Magnusson. An algorithm for checking incomplete proof objects in type theory with localization and unification. In *Types for proofs and programs (Torino, 1995)*, pages 183–200. Springer, Berlin, 1996.

[Mas99] I. A. Mason. Computing with Contexts. *Higher-Order and Symbolic Computation*, 12:171–201, 1999.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Miz] Mizar. `http://web.cs.ualberta.ca:80/~piotr/Mizar/`.

[MN98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[Muñ97] C. Muñoz. Dependent types with explicit substitutions: A meta-theoretical development. In *Proceedings of the International Workshop TYPES '96*, 1997.

[Ned73] R. P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Technische Hogeschool Eindhoven, June 1973.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1994.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.

[Nip93] T. Nipkow. Orthogonal Higher-Order Rewrite Systems are Confluent. In *Proceedings of the International Conference on Typed Lambda Calculi and Application*, pages 306–317, 1993.

[Nup] Proof/Program Refinement Logic. `http://www.cs.cornell.edu/Info/Projects/NuPrl/`.

[Oos94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, March 1994.

[Oos95] V. van Oostrom. Development closed critical pairs. In *Proceedings of the 2nd International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, volume 1074 of *Lecture Notes in Computer Science*, pages 185–200, 1995.

[OR93] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. Technical Report CS-R9361, CWI, 1993. Extended abstract in Proceedings of HOA'93.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[PPM90] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[PVS] The PVS Specification and Verification System. `http://pvs.csl.sri.com/`.

[Raa96] F. van Raamsdonk. *Confluence and normalisation for higher-order rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1996.

[Ran91] A. Ranta. Intuitionistic categorial grammar. *Linguistics and Philosophy*, 14:203–239, 1991.

[Ros96] K. H. Rose. Explicit substitution – tutorial & survey. Lecture Series LS-96-3, BRICS, Department of Computer Science, University of Aarhus, September 1996. v+150 pp.

[San98] D. Sands. Computing with contexts, a simple approach. *Electronic Notes in Theoretical Computer Science*, 10, 1998.

[SP94] P. Severi and E. Poll. Pure type systems with definitions. In *Proceedings of Logical foundations of computer science 1994*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328, 1994.

[SSB99] M. Sato, T. Sakurai, and R. Burstall. Explicit environments. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Application*, pages 340–354, 1999.

[SSK01] M. Sato, T. Sakurai, and Y. Kameyama. A simply typed context calculus with first-class environments. *LNCS*, 2024:359–374, 2001.

[Tai67] W.W. Tait. Intentional interpretations of functionals of finite types. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Tal91] C. L. Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[TD88] A. Troelstra and D. van Daalen. *Constructivism in mathematics, an introduction*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, 1988.

[Ter89] J. Terlouw. Een andere bewijstheoretische analyse van GSTT's. Technical report, Dept. Computer Science, University of Nijmegen, 1989.

[Vri87] R.C. de Vrijer. *Surjective Pairing and Strong Normalisation*. PhD thesis, Universiteit van Amsterdam, January 1987.

[Wol93] D.A. Wolfram. The clausal theory of types. *Cambridge Tracts in Theoretical Computer Science*, 21, 1993.

[WV00] J.B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symp. Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 412–428. Springer-Verlag, 2000.

[Yar] The proof-assistant Yarrow.
`http://www.cs.kun.nl/~janz/yarrow/`.

[Zee89] H. Zeevat. A compositional approach to discourse representation theory. *Linguistics and Philosophy*, 12:95–131, 1989.

[Zuc75] J. Zucker. Formalization of classical mathematics in Automath. In *Colloque International de Logique*, pages 135–145, Clermont-Ferrand, France, 1975. (Colloques Internationaux du Centre National de la Recherche Scientifique).

[Zwa98] J. Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow.* PhD thesis, Eindhoven University of Technology, 1998.

Samenvatting

Dit proefschrift is, in het Nederlands vertaald, getiteld 'Contexten in Lambda Calculus', en het behandelt het formaliseren en automatisch verifiëren van wiskundig redeneren.

Het formaliseren van wiskundig redeneren omvat enerzijds het formaliseren van wiskundige concepten, zoals verzamelingen, relaties en algebra's, en anderzijds het formaliseren van de manier waarop redeneren over dergelijke wiskundige concepten wordt beoefend, zoals het aannemen van axioma's, het vaststellen van definities en het bewijzen van stellingen. In een geformaliseerd wiskundig argument worden de redeneerpatronen onderkend als onafhankelijk van de inhoud van het argument. Door een argument te formaliseren en te verifiëren dat de redeneerstappen toepassingen zijn van zekere consistente axioma's en regels, kunnen eventuele fouten in het argument ontdekt worden. Op deze wijze neemt het vertrouwen in de juistheid van het argument toe. Hier wordt een collectie axioma's en regels consistent genoemd als daaruit geen tegenspraak afgeleid kan worden.

Getypeerde lambda calculi vormen één soort formele systemen waarin wiskundig redeneren kan worden geformaliseerd.

Dit proefschrift draagt een steentje, of beter gezegd, een (context) kubusje bij aan het formaliseren van bijzondere wiskundige structuren in getypeerde lambda calculi op een voor wiskundigen en logici comfortabel niveau van abstractie. Preciezer geformuleerd, in dit proefschrift wordt geïllustreerd hoe wiskundige structuren, bestaande uit een collectie van verzamelingen, relaties, functies en aannames daarover, gerepresenteerd kunnen worden in getypeerde lambda calculi en hoe deze representaties gebruikt kunnen worden.

Deze wiskundige structuren worden gerepresenteerd als contexten. Contexten zijn expressies van getypeerde lambda calculi met 'gaten' waarin andere expressies ingevuld kunnen worden. Contexten zijn in lambda calculi een meta-notie. Representeren van en redeneren over contexten binnen de taal zelf is niet gedefinieerd.

In meer technisch detail behandelt dit proefschrift de volgende onderwerpen. In Hoofdstukken 1 en 2 wordt een algemene introductie gegeven in formele systemen en contexten. In Hoofdstuk 3 wordt eerst een uitbreiding, genoemd de context calculus λc, van ongetypeerde lambda calculus gedefinieerd, ter voorbereiding op de daarop volgende hoofdstukken. In Hoofdstukken 4 en 5 behandelen we contexten van de ongetypeerde lambda calculus, contexten van de simpel getypeerde lambda calculus en De Bruijn's segmenten. We laten zien dat deze gerepresenteerd kunnen

worden in de context calculus λc voorzien van verschillende typeringssystemen. In Hoofdstuk 6 definiëren we de context kubus $\lambda[]$, een collectie van acht getypeerde context calculi, waarin contexten van Barendregt's lambda kubus gerepresenteerd kunnen worden. Hiermee generaliseren we de context calculus λc. We illustreren hoe de bovengenoemde wiskundige structuren in de context kubus $\lambda[]$ gerepresenteerd worden en hoe we met deze representaties kunnen rederenen. We sluiten af door in Hoofdstuk 7 te schetsen hoe de context kubus $\lambda[]$ verder uitgebreid kan worden om de notie van subtyperen te ondervangen. In een dergelijke uitbreiding zouden algemene uitspraken zoals 'iedere $\mathcal{A}$ is $\mathcal{B}$', bijvoorbeeld 'iedere equivalentierelatie is een reflexieve relatie', gerepresenteerd kunnen worden.

De toepassingen van de context calculus λc en de context kubus $\lambda[]$ kunnen terugvertaald worden naar Barendregt's lambda kubus, zoals we dat laten zien in Hoofdstukken 4, 5 en 6. De consequentie hiervan is dat deze formalisering van wiskundige structuren gebruikt kan worden in bestaande *proof checkers*, computer programma's voor het automatisch verifiëren en in eenvoudigere gevallen voor het vinden van bewijzen, die gebaseerd zijn op getypeerde lambda calculi.